Apress®

异步图书
www.epubit.com

精通 ASP.NET Core MVC（第 7 版）

Pro ASP.NET Core MVC 2

Seventh Edition

[美] 亚当 · 弗里曼（Adam Freeman） 著

郝冠军 孙臻 闫小迪 张淯易 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

目录

版权信息

书名：精通ASP.NET Core MVC （第7版）

ISBN：978-7-115-55961-6

施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

版　权

著　　　　[美] 亚当•弗里曼（Adam Freeman）

译　　　　郝冠军　孙　臻　闫小迪　张湄易

责任编辑　谢晓芳

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

本书详细解释ASP.NET Core MVC的架构、功能和应用，深入阐述构建现代可扩展的Web应用程序的工具、技术和方法，揭示如何为.NET Core平台创建轻型的移动端应用程序。本书主要内容包括MVC模式、C#基本特性、Razor、Visual Studio、MVC应用程序的单元测试、实际应用程序的创建、URL路由、高级路由特性、控制器、依赖注入、过滤器、API控制器、视图、视图组件、标签助手、模型绑定、模型验证、ASP.NET Core Identity、模型约定和操作约束等。

本书适合.NET开发人员和 Web开发人员阅读，也可供计算机相关专业的师生阅读。

作者简介



Adam Freeman是一位经验丰富的IT专家，曾在一系列公司担任高级职务，最后的职务是一家全球性银行的首席技术官（Chief Technology Officer，CTO）和首席运营官（Chief Operating Officer，COO）。退休以后，他喜欢写作和长跑。

技术审校者简介

Fabio Claudio Ferracchiati是一名高级咨询师、高级分析师/开发人员，擅长使用微软的技术。他目前就职于BluArancio公司。他是微软认证的.NET解决方案开发人员、微软认证的.NET应用程序开发人员、微软认证的专业人士，还是一位笔耕不辍的作者。在过去十年中，他为意大利和国际杂志撰文，在计算机领域与他人合著了十几本图书。

推荐序

虽然ASP.NET Core MVC 3 已经发布了，但是本书的翻译和出版非常及时，内容非常不错，满足了成千上万希望学习.NET的开发人员对更好的学习资料的强烈需求，感谢本书的译者花时间翻译了一本好书，帮助中国的开发者利用ASP.NET Core MVC构建应用。

本书是ASP.NET方面的畅销书，作者Adam Freeman在本书里对ASP.NET Core MVC进行了详细介绍，说明了如何从ASP.NET Core MVC中获得最大收益。结合具体示例和代码，本书展示了ASP.NET Core MVC的方方面面，解释了如何使用ASP.NET Core MVC构建Web应用程序。

本书围绕ASP.NET Core MVC深入探讨构建可

扩展的Web应用程序所需要的工具和技术，有助于开发人员为.NET平台生成更精简的应用程序。

张善友

微软最有价值专家、腾讯云最有价值专家、华为云最有价值专家

深圳市友浩达科技有限公司CEO

服务与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关服务和支持。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，单击“提交勘误”，输入勘误信息，单击“提交”按钮即可（见下图）。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在

异步社区兑换优惠券、样书或奖品。



与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我

们；有意出版图书的作者也可以到异步社区在线投稿（直接访问www.epubit.com/ contribute即可）。

如果您所在学校、培训机构或企业想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接通过邮件发送给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT图书和相关学习产品，为

作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、人工智能、测试、前端、网络技术等。

异步社区

微信服务号

第一部分　ASP.NET Core MVC

ASP.NET Core MVC对于使用微软平台的Web开发人员来说是一次彻底的转变。它强调清晰的架构、设计模式和可测试性，并且不会试图隐藏Web的工作方式。

本书第一部分旨在介绍MVC开发的基本概念，包括ASP.NET Core MVC中的新功能，并在实践中体验框架的使用方式。

第1章　ASP.NET Core MVC背景

ASP.NET Core MVC是一个来自微软的Web应用程序开发框架，它结合了模型-视图-控制器（MVC）体系结构的有效性和整洁性、敏捷开发的想法和技术，以及.NET平台的最佳部分。在本章中，你将了解微软创建ASP.NET Core MVC的原因，看看它如何与其前身和替代品进行比较。最后，本章将概述ASP.NET Core MVC中的新特性以及本书所涵盖的内容。

1.1　ASP.NET Core MVC的历史

最早的ASP.NET是在2002年推出的，当时微软热衷于保护其在传统桌面应用程序开发中的主导地位，并将互联网视为威胁。图1-1说明了当时出现

的ASP.NET Web Forms技术栈。

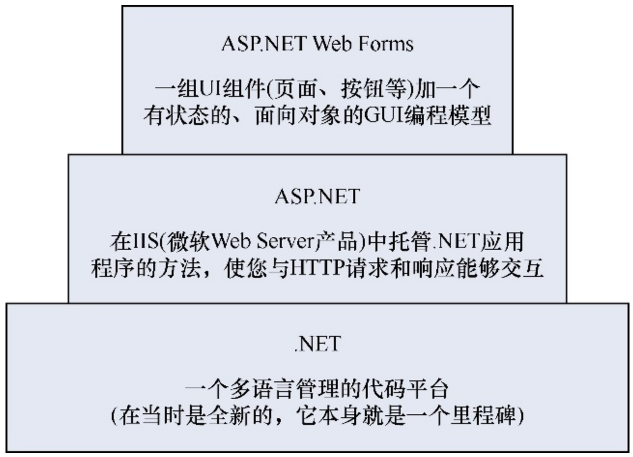


图1-1　ASP.NET Web Forms技术栈

1.1.1　ASP.NET Web Forms

微软试图使用ASP.NET Web Forms将用户界面（User Interface，UI）模拟为服务器端控件对象层，隐藏超文本传输协议HTTP（本身是无状态的）和超文本标记语言HTML（当时许多开发人员对此还不熟悉）。每个控件在请求之间保持自己的状态，在需要时自动渲染为HTML，并将客户端事件（如按钮单击）与服务器端相应的事件处理程序

代码相关联。实际上，Web窗体是一个巨大的抽象层，旨在通过Web传递经典的事件驱动的图形用户界面（Graphics User Interface，GUI）。

其思想是使Web开发的体验与开发桌面应用程序一致。开发人员可以基于有状态的用户界面进行考虑，而无须使用一系列独立的HTTP请求和响应。微软可以使Windows桌面开发人员向新型Web应用程序开发领域实现无缝转型。

ASP.NET Web Forms存在的问题

传统的ASP.NET Web Forms开发从原则上来说曾经很好，但事实证明它更加复杂。

- View State权重：跨请求维护状态的实际机制（称为“View State”）导致在客户端和服务器之间传输大量数据。即使是大小适中的Web应用程序，这些数据也可能会达到几百千字节，并

且每次请求都会来回传递数据，从而导致响应时间更慢，并增加了服务器的带宽需求。

- 页面生命周期：连接客户端事件与服务器端事件处理程序代码（页面生命周期的一部分）的机制可能会变得复杂和棘手。很少有开发人员在不产生View State错误或发现某些事件处理程序莫名失败的情况下，能在运行时成功操纵控件层。
- 关注点分离的错误理念：ASP.NET Web Forms的代码隐藏模型提供了将应用程序代码从HTML标记中移除到单独的代码隐藏类中的方法。这是为了分离逻辑层和表现层，但实际上，又鼓励开发人员将表现层代码（如操纵服务器端控件树）与其应用程序逻辑（如操纵数据库数据）混在这些怪异的后台代码类中。最终的结果可能是难以理解的。
- 对HTML的有限控制：服务器控件将其自身呈现为HTML，但不一定是你想要的HTML。在ASP.NET的早期版本中，HTML输出无法符合Web标准，或者不能很好地利用层叠样式表

（CSS），并且服务器控件生成难以预测且复杂的ID属性，这些属性难以使用JavaScript访问。这些问题在最近的ASP.NET Web Forms版本中有所改进，但是获取你期望的HTML仍然是比较困难的。

- 有漏洞的抽象：ASP.NET Web Forms尽可能隐藏HTML和HTTP。当你尝试实现自定义行为时，你经常会放弃抽象，这迫使你对回发事件机制进行逆向工程，或执行笨拙的操作以使其生成所需的HTML。
- 低可测试性：ASP.NET Web Forms的设计人员无法预料到自动测试将成为软件开发的重要组成部分。他们设计的紧密耦合架构不适合单元测试。集成测试也可能是一个挑战。

ASP.NET Web Forms并非一无是处，实际上，微软为提高标准合规性和简化开发流程付出了巨大的努力，甚至从原始的ASP.NET MVC框架中获取了一些功能，并将其应用于ASP.NET Web Forms。

当你需要快速的结果时，ASP.NET Web Forms表现优异，你可以在一天内拥有一个相当复杂的Web应用程序。但除非你在开发过程中足够小心，否则你会发现你创建的应用程序难以测试和维护。

1.1.2 起初的MVC框架

2007年10月，微软发布了一个基于现有ASP.NET平台的新开发平台，旨在直接回应对ASP.NET Web Forms的批评和竞争平台（如Ruby on Rails）的普及。新平台称为ASP.NET MVC框架，并反映了Web应用程序开发的新兴趋势，如HTML和CSS标准化、REST Web服务、有效的单元测试以及开发人员应该接受HTTP的无状态特性的想法。

支持最初MVC框架的概念现在看起来很自然而且显而易见，但是它们在2007年的.NET Web开

发世界中是缺乏的。ASP.NET MVC框架的引入使微软的Web开发平台重新回到了现代。

MVC框架还表明微软的态度发生了重大变化，微软以前曾试图控制Web应用程序工具链中的每个组件。现在微软基于jQuery等开源工具构建了MVC框架，从竞争（并且更为成功的）平台中获得了设计约定和最佳实践，并将源代码发布到MVC框架，供开发人员审查。

起初的**MVC**框架存在的问题

微软在创建MVC框架时，基于现有的ASP.NET平台，这是有道理的，因为该平台具有很多固有的底层特性，ASP.NET开发人员都熟知和理解这些特性。

但是，将MVC框架移植到最初为ASP.NET

Web Forms设计的平台上是需要妥协的。MVC框架开发人员逐渐喜欢使用配置设置和代码调整，来禁用或重新配置对Web应用程序没有任何影响但对整个程序正常工作来说必需的特性。

随着MVC框架的普及，微软开始将一些核心功能添加到ASP.NET Web Forms中。结果越来越不相匹配，其中需要用来支持MVC框架的独特设计特性被扩展到支持ASP.NET Web Forms，却为了让所有的东西融合在一起而让设计变得更加不相匹配。同时，微软开始使用创建Web服务（Web API）和实时通信（SignalR）的新框架来扩展ASP.NET。新的框架添加了自己的配置和开发约定，每个都有自己的优点和特异之处，结果导致了零乱的混乱状态。

1.2　ASP.NET Core

2015年，微软公布了ASP.NET和MVC框架的新方向，也就是本书要讨论的主题ASP.NET Core MVC。

ASP.NET Core基于.NET Core构建，它是.NET Framework的跨平台版本，没有Windows平台特定的应用程序编程接口（Application Programming Interface，API）。虽然Windows仍然是主要的操作系统，但Web应用程序越来越多地托管在云平台的小型而简单的容器中，并且通过采用跨平台方法，微软扩展了.NET的覆盖面，使得ASP.NET Core应用程序能够部署到更广泛的托管环境中。另外，ASP.NET Core还有一个额外的优点——开发人员可以在Linux系统和macOS上创建ASP.NET Core Web应用程序。

ASP.NET Core是一个全新的框架。它更简单、更容易处理，并且没有来自ASP.NET Web Forms的

遗留问题。另外，由于它基于.NET Core，因此它支持在一系列平台和容器上进行Web应用程序开发。

ASP.NET Core MVC提供了基于新的ASP.NET Core平台构建的初始ASP.NET MVC框架的功能。它集成了以前由Web API提供的功能，能以更自然的方式生成复杂内容，并且使关键开发任务（如单元测试）更简单。

1.2.1 ASP.NET Core MVC 2的新特性

ASP.NET Core MVC 2的发布版本重点关注整合，它在早期版本中引入了一些工具和平台变更。ASP.NET Core MVC 2需要.NET Core 2，它具有很多扩展的API，现已在其他Linux发行版上受到支持。有用的变化包括一个新的元数据包系统（它简化了NuGet包的管理），一个新的ASP.NET Core配

置系统，以及对Entity Framework Core 2的支持。最重要的新功能是Razor Pages，它尝试重新构建应用，并使用更现代的平台创建与Web Pages相关的开发风格，但MVC开发人员对Razor Pages并不感兴趣（因而在本书中没有描述）。

1.2.2　ASP.NET Core MVC的主要优点

本节将简要介绍新的MVC平台如何克服ASP.NET Web Forms和初始MVC框架的遗留问题，以及如何完善ASP.NET。

1．MVC架构

ASP.NET Core MVC遵循称为模型-视图-控制器（MVC）的模式，指导ASP.NET Web应用程序及其包含的组件之间的交互。

区分MVC架构模式和ASP.NET Core MVC实现

很重要。MVC模式并不新颖，它可以追溯到1978年Xerox PARC的Smalltalk项目，但是它作为Web应用程序的一种开发模式，已经得到了广泛普及，原因如下。

- 用户与遵守MVC模式的应用程序进行交互是遵循自然循环的。用户执行一个动作，应用程序进行响应，更改其数据模型并向用户传递更新的视图。然后重复这一循环。这对作为一系列HTTP请求和响应来传输的Web应用程序来说非常合适。
- Web应用程序需要组合几种技术（例如数据库、HTML和可执行代码等），通常分为一系列层次。这些组合产生的模式会自然地映射到MVC模式的概念上。

ASP.NET Core MVC实现了MVC模式，与ASP.NET Web Forms相比，ASP.NET Core MVC极大改善了关注点的分离。实际上，ASP.NET Core

MVC实现了MVC模式的变体，特别适用于Web应用程序。你将在第3章中更多地了解此架构的理论和实践。

2．可扩展性

ASP.NET Core和ASP.NET Core MVC已构建为一系列具有明确特征的独立组件，能满足.NET接口的需求，也可构建在抽象基类上。你可以轻松地用自己的实现替换关键组件。一般来说，ASP.NET Core MVC为每个组件提供了以下3个选项。

- 使用组件的默认实现（对于大多数应用程序来说应该是足够的）。
- 从默认实现派生一个子类来调整其行为。
- 使用接口或抽象基类的新实现完全替换组件。

你将从第14章开始，了解各种组件、如何替换以及为什么要调整或替换每个组件。

3．严格控制HTML和HTTP

ASP.NET Core MVC能够生成清晰、符合标准的标签。它的内置标签帮助器能产生符合标准的输出，但与ASP.NET Web Forms相比，有更重要的理念上的变化。ASP.NET Core MVC并不会生成一些难以控制的HTML控件，而是鼓励你创建简单而优雅的标签，并使用CSS进行样式化。

当然，如果你想要为诸如日期选择器或级联菜单之类的复杂UI元素使用一些现成的小部件，那么ASP.NET Core MVC采用的“无特定要求”方法可以很轻松地使用各种最佳组合的客户端库，如jQuery、Angular或Bootstrap CSS库。ASP.NET Core MVC与这些库相互配合得很好，微软已包含这些模板以启动新的开发项目。

ASP.NET Core MVC与HTTP协调工作。你可以

控制在浏览器和服务器之间传递的请求，因此你可以根据需要调整用户体验。使用 Ajax 更加容易，创建 Web 服务来接收浏览器 HTTP 请求是一个简单的过程。

4．可测试性

ASP.NET Core MVC架构在使应用程序变得可维护和可测试方面提供了良好的开端，因为你可以将不同的应用程序关注点自然地分离成独立的部分。此外，ASP.NET Core平台和ASP.NET Core MVC框架的每个部分都可以为单元测试进行隔离和替换，可以使用任何流行的开源测试框架（如xUnit）。

在本书中，你将看到如何为ASP.NET MVC控制器编写整洁、简单的单元测试示例。为了模拟各种场景，这些示例使用各种测试和模拟策略来支持

框架组件的虚构或模拟实现。即使你以前从来没有写过单元测试，这也是一个很好的开始。

可测试性不仅仅是单元测试的问题。ASP.NET Core MVC应用程序也可以与UI自动化测试工具一起使用。你可以编写模拟用户交互的测试脚本，而不需要猜测框架将生成哪些HTML元素结构、CSS类或ID，你不必担心页面结构发生意外的变化。

5. 强大的路由系统

统一资源定位器（URL）的风格随着Web应用技术的发展而发展，比如，以下URL越来越少见。

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

取而代之的是一种更简单、更干净的格式：

```
/to-rent/chicago/2303-silver-street
```

采用这种URL结构有一些很好的理由。第一，搜索引擎会对URL中找到的关键字加权。搜索“rent in Chicago”（芝加哥租房）更有可能找到更简单的网址。第二，许多网络用户现在已经足够了解URL，并乐于通过在浏览器的地址栏中键入导航选项。第三，当人们理解URL的结构时，才更有可能链接它，与朋友分享，甚至通过手机朗读。第四，它不会向公共Internet暴露你的应用程序的技术细节、文件夹和文件名结构，因此你可以自由地更改底层的实现，而不会破坏所有的传入链接。

整洁的URL在早期的框架中很难实现，但ASP.NET Core MVC默认使用称为URL路由的功能来提供整洁的URL。这样可以控制你的URL模式及其与应用程序之间的关系，自由地为用户创建有意义和有用的URL模式，而无须遵守预定义模式。当然，这意味着你可以轻松地定义一种现代REST风

格的URL模式。

6．现代**API**

微软的.NET平台随每个主版本的发展而发展，支持甚至定义了现代编程的最新方向。ASP.NET Core MVC是为.NET Core构建的，因此其API可以充分利用C#程序员熟悉的语言和运行时创新，包括await关键字、扩展方法、lambda表达式、匿名和动态类型以及语言集成查询（LINQ）。

许多ASP.NET Core MVC API方法和编码模式与早期平台相比遵循更整洁、更具表现力的方式。不要担心，如果你尚不了解最新的C#语言特性，第4章会提供MVC开发中最重要的C#特性总结。

7．跨平台

以前的ASP.NET版本特定于Windows系统，需

要使用Windows桌面编写Web应用程序，使用Windows服务器才能部署和运行它们。微软使ASP.NET Core支持跨平台，包括开发和部署。ASP.NET Core可用于不同的平台，包括macOS和一系列流行的Linux发行版。跨平台支持使得部署ASP.NET Core MVC应用程序变得更加容易，并且可以很好地支持应用程序容器平台，如Docker等。

当前大多数ASP.NET Core MVC开发很可能会使用Visual Studio完成，但微软也创建了一个名为Visual Studio Code的跨平台开发工具，这意味着ASP.NET Core MVC开发不再局限于Windows平台。

8．ASP.NET Core MVC是开源的

与以前的Microsoft Web开发平台不同，你可以免费下载ASP.NET Core和ASP.NET Core MVC的源

代码，甚至修改和编译自己的版本。当你要调试跟踪进入一个系统组件并希望进入其代码内部（甚至阅读原始的程序员注释）时，这是非常有价值的。如果你正在构建一个高级组件，并希望了解进一步开发的可能性，或想了解内置组件如何进行实际工作，这也是非常有用的。

你可以从GitHub下载ASP.NET Core和ASP.NET Core MVC的源代码。

1.3　预备知识

为了充分利用本书，你应该熟悉Web开发的基础知识，了解HTML和CSS的工作原理，并掌握C#的相关知识。如果你对客户端细节（如JavaScript）有些模糊，请不要担心。重点是本书中的服务器端开发，你可以通过示例获取所需的内容。第4章将总结MVC开发中最有用的C#语言特性，如果你正

在从早期版本转到最新的.NET版本，你将发现它们非常有用。

1.4　本书的结构

本书分为两部分，每一部分都涵盖了一系列相关主题。

本书第一部分将从ASP.NET Core MVC的背景开始，解释MVC模式的优点和实际影响，介绍ASP.NET Core MVC的功能，并描述每个ASP.NET Core MVC程序员需要学习的工具和C#语言功能。

在第2章中，你将通过创建一个简单的Web应用程序，深入了解主要组件、构建块以及它们如何组合在一起。然而，本书第一部分主要介绍如何开发一个名为SportsStore的项目，通过该项目，展示从开始到部署的实际开发流程，并介绍ASP.NET

Core MVC的主要特性。

本书第二部分将解释用于构建SportsStore应用程序的ASP.NET Core MVC功能的内部工作原理。该部分将展示每个功能如何工作，解释它所扮演的角色，并展示可用的配置和自定义选项。本书第一部分介绍广泛的背景基础，第二部分则深入讨论细节。

1.5　如何获取本书的示例代码

在GitHub网站上搜索“pro-asp.net-core-mvc-2”，即可下载本书所有章节的示例代码。下载的内容不需要修改，并包含所有必要的资源。

1.6　联系作者

如果你在使用本书的示例代码时遇到问题，或者如果你在本书中发现问题，你可以通过电子邮件

adam@adam- freeman.com联系作者，作者会尽力帮助你。

1.7 小结

本章介绍了ASP.NET Core MVC的背景以及它是如何从ASP.NET Web Forms和起始的ASP.NET MVC框架发展而来的，阐述了使用ASP.NET Core MVC的好处以及本书的结构。在下一章中，你将在一个简单的示例程序中看到ASP.NET Core MVC先进的特性。

第2章　首个MVC应用程序

鉴赏一款软件开发框架的最佳方式就是深入其中并使用它。在本章中，你将使用ASP.NET Core MVC创建一个简单的数据输入应用程序。整个开发过程可分解为多个小的步骤，你将了解到清晰的MVC应用程序的构造。当然为了简化，也会跳过一些技术细节。但别担心，如果你是MVC新人，你会发现许多感兴趣的内容。对于使用但未解释的部分，本章也会提供参考引用，以便于你发现所有的细节。

本书的内容更新

微软对.NET Core和ASP.NET Core MVC有一项

活跃的开发计划，这意味着在你阅读本书时可能会有更新的版本可用。要求读者每隔几个月就买一本新书是不现实的，尤其是在大多数改动相对较小的情况下。因此，作者将免费更新本书，并将内容放在GitHub上，以应对由次要版本引起的更改。这种更新对于作者来说是一个实验（对于 Apress 来说也是），作者还不知道这些更新可能采取什么形式，尤其是因为作者不知道未来主要版本的ASP.NET Core MVC将包含什么，但目标是通过补充包含的示例来延长本书的寿命。作者无法承诺这些更新会是什么样的，它们会采取什么形式，或者在将它们添加到本书的新版本之前，作者会花费多长时间。当新的ASP.NET Core MVC版本发布时，请保持开放的心态并检查本书的GitHub主页。如果你对如何改进更新有想法，请发电子邮件至adam@adam-freeman.com。

2.1　安装Visual Studio

本书使用的集成开发环境是Visual Studio 2017，它提供了使用ASP.NET Core MVC时所需的各种功能。本书中的示例使用的是免费的Visual Studio 2017社区版本，有需要的读者可以从Visual Studio官网下载。当你安装Visual Studio时，你必须选择.NET Core cross-platform development选项，如图2-1所示。

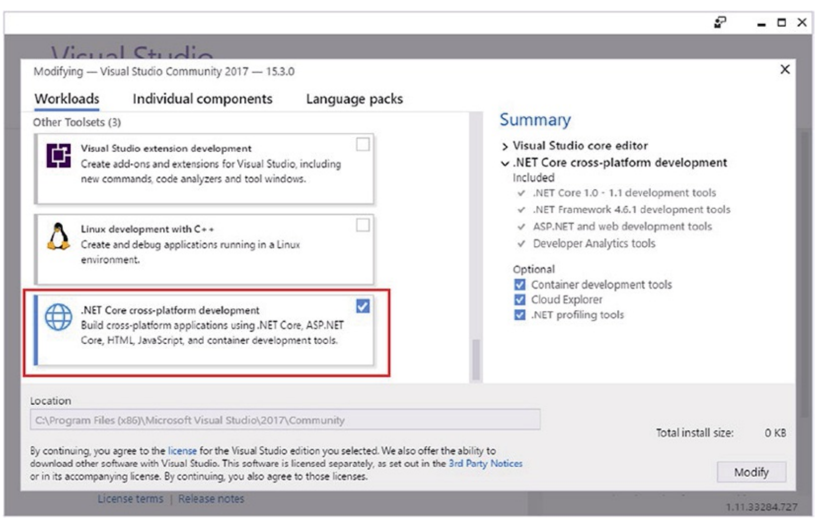


图2-1　选择.NET Core cross-platform development选项

注　意

Visual Studio 2017早于ASP.NET Core MVC 2发布。如果你已为早期版本的ASP.NET Core MVC安装了Visual Studio，则必须使用最新的Visual Studio更新。你可以运行Visual Studio安装程序，并为正在使用的Visual Studio版本选择Update选项。

提　示

Visual Studio只支持Windows。你可以在其他平台上创建ASP.NET Core MVC应用程序并使用本书Visual Studio示例中的代码，但是其他平台可能不包含本书示例中使用的工具。

详情可参见第13章。

2.2　安装.NET Core 2.0 SDK

Visual Studio安装包含了ASP.NET Core MVC开发所需的所有功能，但不包括必须单独下载和安装的.NET Core 2.0。

访问微软官网，下载并运行适用于Windows的.NET Core SDK安装程序。执行完安装程序后，

打开新的命令提示符或PowerShell窗口并运行以下命令以显示已安装的.NET版本：

```
dotnet --version
```

2.3 创建新的ASP.NET Core MVC项目

本节从创建一个新的ASP.NET Core MVC项目开始讲述。在左侧窗格中从File菜单中选择New→Project菜单来打开New Project对话框。如果你选择Installed→Visual C#→Web，你将看到ASP.NET Core Web Application（.NET Core）项目模板，按照图2-2选择项目类型。

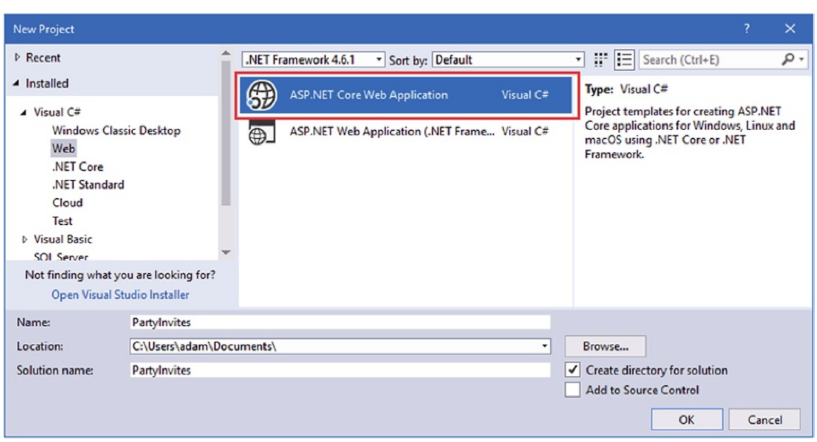


图2-2　选择ASP.NET Core Web Application项目模板

提　示

项目模板的选择可能会令人困惑，因为它们的名称非常相似。ASP.NET Web Application（.NET Framework）模板会使用ASP.NET和MVC框架的遗留版本创建项目，这一框架早于ASP.NET Core。另外两个模板用于创建ASP.NET Core应用程序，它们在运行时会有不同，原则上你可以任意选择.NET Framework或.NET Core选项，第6章会解释它们之间的区别。但是请注意，本书使用了.NET Core选项，所以请你尽量也选择这个选项以确保能获得和示例代码相同的结果。

将新项目的名称设置为PartyInvites，单击OK按钮继续，你将看到另一个对话框，要求你为项目设置初始内容。确保从下拉菜单中选择.NET Core和ASP.NET Core 2.0以配置初始项目，如图2-3所示。

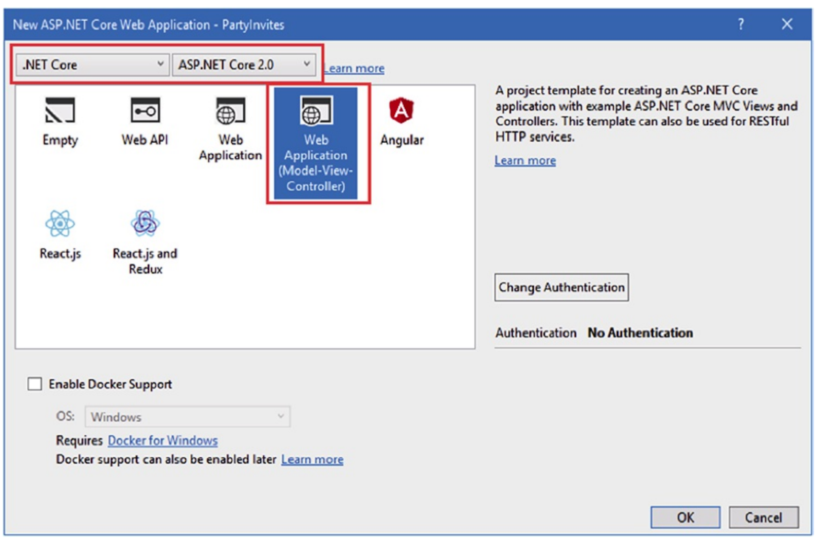


图2-3　进行初始项目配置

这里有几个模板选项，每个选项都会创建一个具有不同起始内容的项目。在本章中，选择Web

Application（Model-View-Controller）选项，该选项会设置MVC应用程序使用预定义的内容启动开发。

注　意

只有本章使用Web Application（Model-View-Controller）项目模板。作者不喜欢使用预定义的项目模板，因为它们鼓励开发人员将一些重要的特性（如身份验证）视为黑盒。本书的目标是让你了解和管理MVC应用程序的各个方面，所以本书的其余部分均使用了空模板。这一章是关于快速入门的，所以使用Web Application（Model-View-Controller）模板非常合适。

单击Change Authentication按钮，并确保选中了No Authentication单选按钮，进行身份验证设置，如图2-4所示。这个项目不需要任何身份验证，但是第28～30章解释了如何对ASP.NET应用程序进行安全保护。

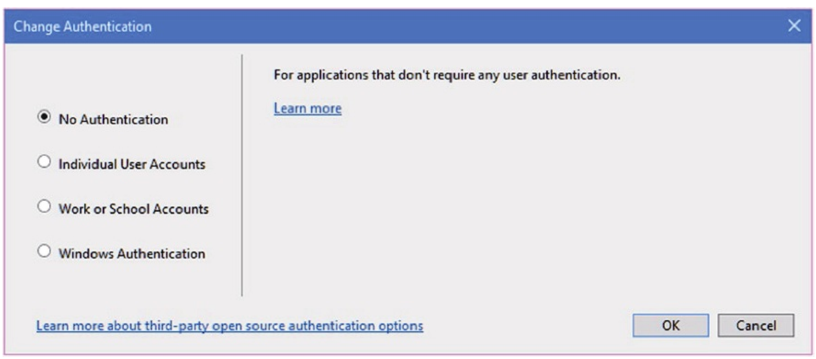


图2-4　进行身份验证设置

单击OK按钮，然后关闭Change Authentication对话框。确保没有勾选Host in the Cloud复选框，然后单击OK按钮创建PartyInvites项目。

在Visual Studio中创建了这个项目后，你将看到Solution Explorer窗格中显示了许多文件和文件

夹，如图2-5所示。这是使用Web Application模板创建的新MVC项目的默认项目目录，之后你将快速了解Visual Studio创建的每个文件和文件夹的用途。

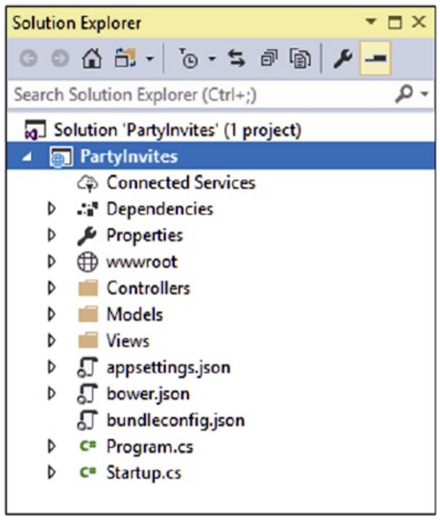


图2-5　ASP.NET Core MVC项目的初始文件与文件夹结构

提　示

如果你看到的是Pages文件夹而不是

Controllers、Models和Views文件夹，那么说明你选择的是Web Application模板而不是（相似而令人困惑的）Web Application（Model-View-Controller）模板。你必须删除你创建的项目并重新开始。

你可以通过从Debug菜单中选择Start Debugging来调试运行应用程序（如果提示你启用调试，只需要单击OK按钮）。当你这样做时，Visual Studio会编译应用程序，使用名为IIS Express的应用程序服务来运行它，并打开Web浏览器来请求应用程序的内容。你可以在图2-6中看到结果。

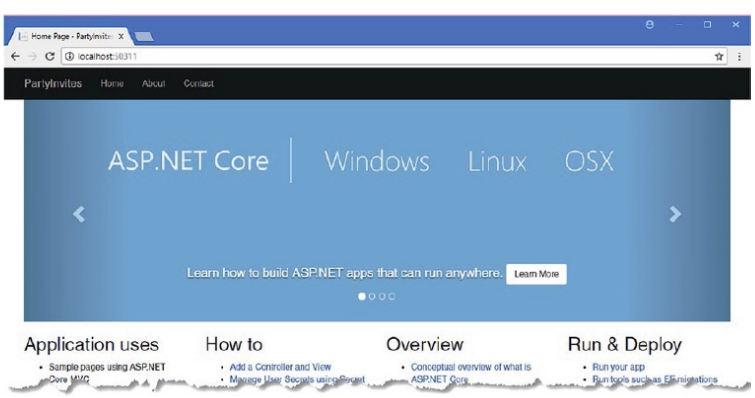


图2-6　运行示例项目

当Visual Studio使用Web Application（Model-View-Controller）模板创建项目时，它会添加一些基本的代码和内容，这就是你在运行应用程序时所看到的内容。在本章的其他部分，将替换这些内容，以创建一个简单的MVC应用程序。

当你完成的时候，请通过关闭浏览器窗口停止调试，或者返回Visual Studio，并从Debug菜单中选择Stop Debugging以停止调试。

Visual Studio通过打开浏览器来显示项目。选择你安装的任何浏览器，单击IIS Express工具栏按钮右边的箭头，从Web Browser菜单的选项列表中选择要使用的浏览器，如图2-7所示。

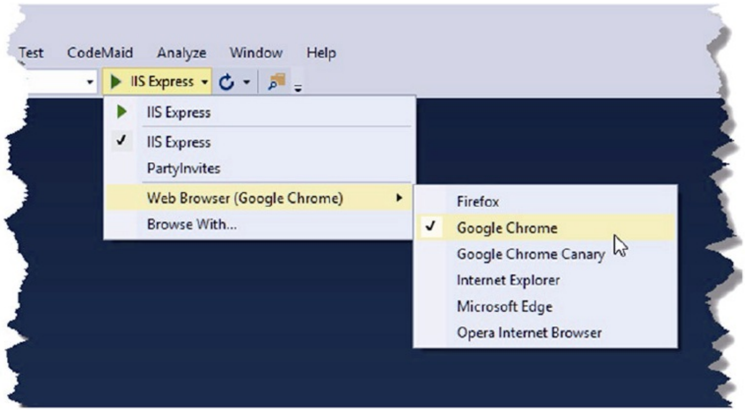


图2-7　选择浏览器

本书的所有截图将使用谷歌Chrome或谷歌Chrome金丝雀版浏览器，但你也可以使用任何别的浏览器（包括Microsoft Edge和最新版本的Internet Explorer）来显示书中的示例。

2.3.1　添加控制器

在MVC模式中，传入的请求由控制器来处理。在ASP.NET Core MVC中，控制器是一个C#文件（通常从Microsoft.AspNetCore.Mvc.Controller类继承，这个类已内置于MVC控制器的基本类中）。

控制器中的每个公共方法都称为操作（action）方法，这意味着你可以通过Web基于某种URL来调用执行操作方法。MVC约定通常会将控制器放置在Controllers文件夹中，当创建项目时，Visual Studio会自动创建这个文件夹。

提　示

你不需要一定遵循这个或大多数其他的MVC约定，但是建议你仍然遵守——因为它将帮助你理解本书中的示例。

Visual Studio为项目添加了一个默认的控制器

类，如果你在Solution Explorer中展开Controllers文件夹，就可以看到它。这个文件被命名为HomeController.cs。控制器类的名称的后半部分包含Controller这个单词，这意味着当你看到一个名为HomeController.cs的文件时，就知道它包含一个名为Home的控制器，Home是MVC应用中使用的默认控制器。单击Solution Explorer中的HomeController.cs文件，通过Visual Studio打开它并进行编辑，你将看到代码清单2-1所示的C#代码。

代码清单2-1　Controllers文件夹中HomeController.cs文件的初始内容

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
```

```
            return View();
        }

        public IActionResult About() {
            ViewData["Message"] = "Your application des
cription page.";

            return View();
        }

        public IActionResult Contact() {
            ViewData["Message"] = "Your contact page.";

            return View();
        }

        public IActionResult Error() {
            return View(new ErrorViewModel { RequestId
= Activity.Current?.Id
                ?? HttpContext.TraceIdentifier });
        }
    }
}
```

将HomeController.cs中的代码改为代码清单2-2所示的样子：只保留一个方法并且删除其他的方法，更改结果类型及其实现，同时删除没有使用的命名空间所在的using语句。

代码清单2-2　更改Controller文件夹中的HomeController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public string Index() {
            return "Hello World";
        }
    }
}
```

这些修改没有什么特别的效果，但可以作为很好的演示。这里更改了名为Index的方法，以返回字符串“Hello World”。接下来，从Visual Studio的Debug菜单中选择Start Debugging以再次运行这个项目。

提　示

如果之前的应用程序还在运行，请从Debug菜单中选择Restart，或者选择Stop Debugging后再选择Start Debugging，开始这个新的应用程序的调试。

浏览器将向服务器发出一个HTTP请求。默认的MVC配置意味着这个请求将使用Index方法（这是一个操作方法）来处理，并且方法的返回结果将被发送回浏览器，如图2-8所示。

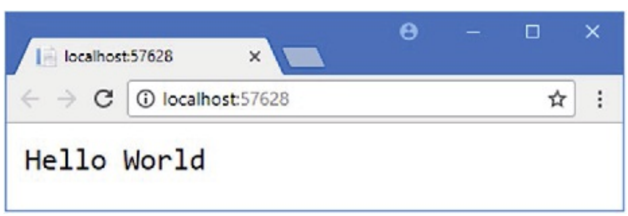


图2-8　操作方法的输出

提　示

请注意，Visual Studio已将浏览器定向到57628端口。你的浏览器在请求这一URL时，返回的很可能是一个不同的端口号，因为Visual Studio在创建项目时会分配一个随机的端口号。如果查看Windows任务栏的通知区域，你将找到IIS Express图标。这是完整的IIS应用服务器的一个简化版本，它包含在Visual Studio中，用于在开发过程中发布ASP.NET Core的内容和服务。第12章将展示如何将MVC项目部署到生产环境中。

2.3.2　理解路由

除了模型、视图和控制器之外，MVC应用程

序还使用了ASP.NET路由系统，它决定了如何把URL映射到控制器和action上。路由是用来决定如何处理请求规则的。当使用Visual Studio创建MVC项目时会自动添加一些默认的路由。你可以请求以下任何一个URL，它们将直接指向HomeController的Index操作方法。

- /。
- /Home。
- /Home/Index。

因此，当浏览器请求http://yoursite/或http://yoursite/Home时，它将从HomeController的Index方法返回输出。你可以通过在浏览器中更改URL来测试一下效果。此时URL会是http://localhost:57628/（你的端口号部分可能与此处的不同），如果你将/Home或/Home/Index添加到URL并按下Enter键，你看到的应该是MVC应用程

序输出“Hello World”。

这是关于ASP.NET Core MVC约定的一个很好的示例。在本例中，我们约定项目中存在一个名为HomeController的控制器，并且该控制器为渲染首页的默认控制器。在此假设Visual Studio为新项目创建的默认路由配置也遵循这个约定。由于确实遵循了这个约定，因此这里自动得到了前面列表中URL的支持。如果没有遵循该约定，则需要修改配置，以指向此处创建的其他控制器。由于这只是一个简单的小示例，因此采用默认配置即可。

2.4　渲染页面

上一个示例的输出并不是HTML，而只是字符串“Hello World”。要为浏览器请求生成HTML响应，我们需要视图（view），视图将告诉MVC如何为浏览器生成请求的响应。

2.4.1　创建并渲染视图

我们需要做的第一件事是修改Index的操作方法，如代码清单2-3所示。更改部分以粗体显示，这是本书遵循的惯例，以使示例更容易理解。

代码清单2-3　修改控制器以在HomeController.cs中呈现视图

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            return View("MyView");
        }
    }
}
```

当从一个操作方法返回一个ViewResult对象时，就是在指示MVC去渲染一个视图。可通过调用View方法来创建ViewResult，并指定要使用的视图，比如MyView。如果运行该应用程序，你可以

看到 MVC会试图查找所要使用的视图，参见图2-9所示的错误消息。

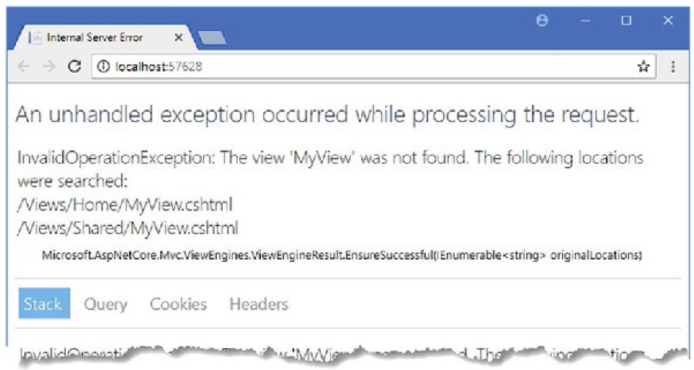


图2-9　错误消息

错误消息对你是很有帮助的，它不仅解释了MVC无法找到上面为操作方法指定的视图，而且显示了在哪些地方进行了查找。视图存储在Views文件夹中，并将其组织为子文件夹。例如，与Home控制器相关联的视图存储在一个名为Views/Home的文件夹中。不是特定于单个控制器的视图则被存储在一个名为Views/Shared的文件夹中。当使用Web Application（Model-View-Controller）模板时，Visual Studio会自动创建Home和Shared文件夹，并

放入一些占位用的视图。

下面开始创建视图，在Solution Explorer中右击Views→Home文件夹并从弹出的菜单中选择Add→New Item。Visual Studio将为你提供一个项目模板列表。使用左侧窗格选择ASP.NET的类别，然后在中央窗格中选择MVC View Page选项，创建视图，如图2-10所示。不要使用RazorPage模板，它与MVC框架无关。

提　示

你将在Views文件夹中看到一些已经存在的文件，这些文件被Visual Studio添加到项目中以提供一些初始内容，其中一些你已经在图2-7中看到。

你可以忽略这些文件。

在Views/Home文件夹中创建一个名为MyView.cshtml的视图文件。Visual Studio将自动创建Views/ Home/MyView.cshtml文件并打开它，进入编辑页面。视图文件的初始内容只是一些注释和占位符。请将它们替换为代码清单2-4所示的内容。

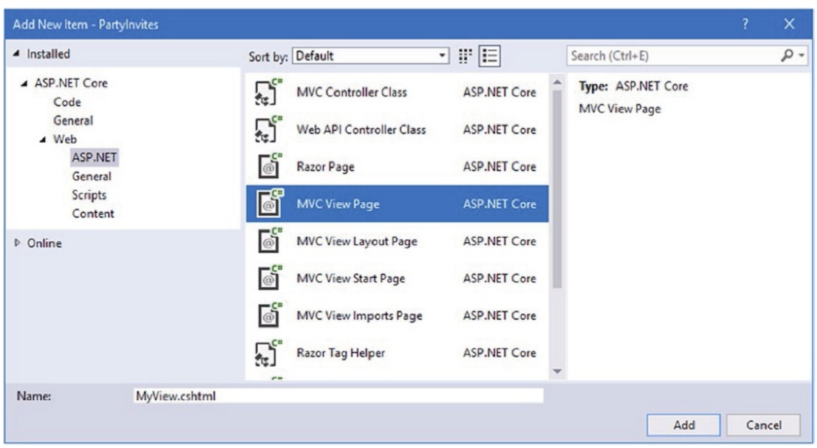


图2-10　创建视图

代码清单2-4　用如下代码替换Views/Home文件夹下MyView.cshtml文件的内容

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
    <div>
        Hello World (from the view)
    </div>
</body>
</html>
The new contents of the view file are mostly HTML. The
exception is the part that looks like this:
...
@{
    Layout = null;
}
...
```

提　示

初学者很容易在错误的文件夹中创建视图文件。如果你没有在Views/Home文件夹中创建一个名为MyView.cshtml的文件，请删除你创建的文件并再次尝试。

这是一个通过Razor视图引擎进行解释的表达式（expression），它处理视图的内容并生成发送到浏览器的HTML。这是一个简单的Razor表达式，它告诉Razor选择不使用布局，它就像HTML模板，将被发送到浏览器（将在第5章中详述）。现在我们先暂时不讨论Razor的问题，稍后回过头来再讨论。我们现在想要查看视图的创建效果，请从Debug菜单中选择Start Debugging以运行应用程序。你应该会看到图2-11所示的结果。



图2-11　测试视图的结果

在最初编辑Index的操作方法时，返回的是一个字符串值。这意味着MVC除了将字符串值传递给浏览器之外什么都没做。现在，Index方法返回了一个ViewResult，MVC渲染了一个视图并返回它所生成的HTML。因为已经告诉MVC应该使用哪个视图，所以MVC会使用命名约定来自动找到它。我们约定视图具有操作方法的名称，并且包含在一个以控制器命名的文件夹/Views/Home/MyView.cshtml中。

除了字符串和ViewResult对象外，还可以从操作方法返回其他结果。例如，如果返回一个RedirectResult，浏览器将被重定向到另一个URL。

如果返回一个HttpUnauthorizedResult，则会强制用户登录。这些对象统称为操作结果。Action Result系统允许你封装和重用操作中的常见响应。第17章将对其做更多的介绍并演示一些不同的方法。

2.4.2 添加动态输出

Web应用程序平台的核心要点就是构造和显示动态输出。在MVC中，控制器的任务是构造一些数据并将其传递给视图，视图再负责将其渲染成HTML。

从控制器传递数据到视图的一种方法是使用ViewBag对象，它是控制器基本类里的一个成员。ViewBag是一个动态对象，可以为它分配任意属性，使这些值在随后渲染的任何视图中都保持可用。代码清单 2-5演示了在HomeController.cs文件中如何通过这种方式传递一些简单的动态数据。

代码清单2-5　在Controllers文件夹下的HomeController.cs文件中传递动态数据

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }
    }
}
```

为ViewBag.Greeting属性赋值，便是为视图提供数据。ViewBag.Greeting属性直到被赋值时才会形成，你不需要提前定义类，这使得你可以更加自如方便地将数据从控制器传递给视图。在视图中再次检索ViewBag. Greeting属性可以获取其数据值，如代码清单2-6所示，它显示了对MyView.cshtml文件所做的相应更改。

代码清单2-6　在Views/Home文件夹下的MyView.cshtml文件中对ViewBag.Greeting的数据值进行检索

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
    </div>
</body>
</html>
```

我们在代码清单2-6中添加的是一个Razor表达式，当MVC使用视图生成响应时，将对其进行求值。当在控制器的Index方法中调用View方法时，MVC定位了MyView.cshtml视图文件并要求Razor视图引擎解析文件的内容。Razor 会寻找类似于本例

中添加的这种表达式并处理它们。在本例中，只要处理这个Razor表达式，就会在视图中插入你在操作方法里赋给ViewBag.Greeting的值。

属性名ViewBag.Greeting只是一个普通的代号，你可以用任何属性名替换它而不会改变其工作方式，只需要保证在控制器中使用的名称与在视图中使用的名称匹配即可。你可以将值分配给多个属性来实现多个数据值从控制器到视图的传递。请通过启动项目来看看这些更改的效果，来自MVC的动态响应如图2-12所示。



图2-12　来自MVC的动态响应

2.5　创建一个简单的数据录入程序

本章剩下的部分将通过构建一个简单的数据录入程序探索 MVC 更多的基本特性。本节将加快进度以演示MVC的实际操作，因此将跳过一些关于背后工作原理的解释。但别担心，后面的章节将深入讨论这些主题。

2.5.1 设置场景

假设有个朋友决定举行一场新年夜派对，她邀请你为此创建一个Web应用程序，以便受邀人能够进行电子回复。要求如下。

- 显示有关聚会信息的主页。
- 可用于回复的表单。
- 实现电子回复表单（RSVP）的验证，将显示感谢页面。
- 总结页面，显示谁会来参加聚会。

下面将介绍如何修改完善在本章开始时创建的

MVC项目，并添加上述几个要求。通过前面介绍的内容可以实现第一个要求，并添加一些HTML到现有视图中，以提供有关聚会的详细信息。代码清单2-7显示了对Views/Home/MyView.cshtml文件所做的修改。

代码清单2-7　在MyView.cshtml文件中显示聚会的详细信息

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
    <div>
       @ViewBag.Greeting World (from the view)
       <p>We're going to have an exciting party.<br />
       (To do: sell it better. Add pictures or somethin
g.)
       </p>
    </div>
</body>
```

```
</html>
```

运行应用程序，通过从Debug菜单中选择Start Debugging，你将看到聚会的详细信息（详细信息用一个占位符表示，你只要懂得操作方式就好），如图2-13所示。

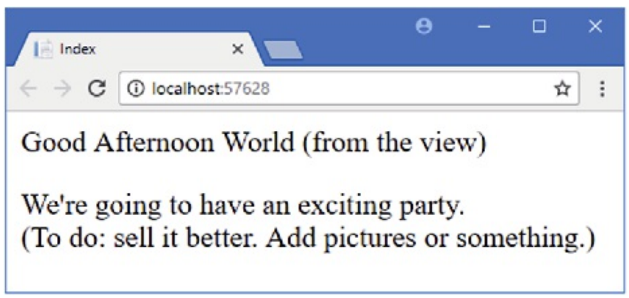


图2-13　添加到视图中的HTML表示聚会的详细信息

2.5.2　设计数据模型

在MVC中，M代表模型，它是应用程序中最重要的部分。模型是现实世界中实际对象、过程和规则的表示。模型通常称为领域模型（Domain Model），包含应用程序域中要建立的C#领域对象（Domain Object）和操作它们的方法。视图和控制

器以一致的方式向客户端暴露领域对象，而设计良好的MVC应用程序往往从设计良好的模型开始，这是作为控制器和视图添加的关键点。

这里只是朋友间使用的一个简单程序，并不需要复杂的PartyInvites项目模型。接下来，将会创建一个名为GuestResponse的领域类，它负责存储、验证和确认电子回复。

MVC约定将组成模型的类放置在名为Models的文件夹中。要创建Models文件夹，可右击PartyInvites项目（项目中已包含Controllers和Views文件夹），从弹出菜单中选择Add→New Folder，并设置文件夹的名称为Models。

为了创建类文件，在Solution Explorer中右击Models文件夹并从弹出菜单中选择Add→Class。将新类的名称设置为GuestResponse.cs并单击Add按

钮。编辑这个新的类文件的内容，如代码清单2-8所示。

代码清单2-8　在Models文件夹下的GuestResponse.cs文件中定义GuestResponse领域类

```
namespace PartyInvites.Models {

    public class GuestResponse {

        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

提　示

你可能已经注意到，WillAttend属性是一个可

以为空的bool值，这意味着可以是true、false或null。2.5.8节会解释这一点。

2.5.3　创建第二个操作和强类型视图

这个应用程序的目标之一是包含一个电子回复表单，因此需要定义一个可以接收请求的操作方法。单个控制器类可以定义多个操作方法，而约定是将相关的操作集中在同一个控制器中。代码清单2-9显示了如何向主控制器中添加一个新的操作方法。

代码清单2-9　在Controllers文件夹下的HomeController.cs文件中添加一个操作方法

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {
```

```
    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        public ViewResult RsvpForm() {
            return View();
        }
    }
}
```

电子回复表单的操作方法是一种不带参数调用的视图方法，它告诉MVC渲染与操作方法关联的默认视图，默认视图与操作方法的名称需要相同，在本例中为RsvpForm.cshtml。

右击Views/Home文件夹并从弹出菜单选择Add→New Item。从ASP.NET类别中选择MVC View Page模板，将新文件命名为RsvpForm.cshtml。然后单击Add按钮创建文件。更改文件的内容，如代码清单2-10所示。

代码清单2-10　更改Views/Home文件夹中RsvpForm.cshtml文件的内容

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RsvpForm</title>
</head>
<body>
    <div>
        This is the RsvpForm.cshtml View
    </div>
</body>
</html>
```

虽然内容主要是HTML代码，但添加了@model Razor表达式，用于创建强类型视图。强类型视图用于渲染特定类型的模型，如果指定要处理的类型（比如PartyInvites.Models中的GuestResponse类），

MVC可以创建一些便于使用这种类型的方法（MVC本身提供了一些便捷的方法，可以在视图中方便地使用这种类型）。

要测试新的操作方法及其视图，可以从Debug菜单中选择Start Debugging来启动应用程序，并使用浏览器导航到/Home/RsvpForm这个URL。

MVC将使用前面描述的命名约定来将请求引导到由主控制器定义的RsvpForm这个操作方法。RsvpForm方法会告诉MVC渲染默认视图，默认视图使用另一个应用程序命名约定来渲染 Views/Home文件夹中的RsvpForm.cshtml文件。图2-14显示了结果。

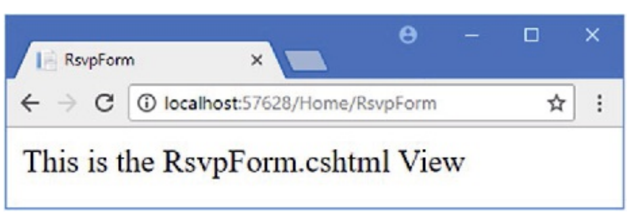


图2-14　渲染第二个视图的结果

2.5.4　链接操作方法

如果从MyView视图中创建一个链接，这样访客就可以看到RsvpForm视图，而不必知道针对特定操作方法的URL，具体操作如代码清单2-11所示。

代码清单2-11　在MyView.cshtml文件中向RSVP Form视图添加一个链接

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
    <div>
         @ViewBag.Greeting World (from the view)
        <p>We're going to have an exciting party.<br />
        (To do: sell it better. Add pictures or somethi
ng.)
        </p>
         <a asp-action="RsvpForm">RSVP Now</a>
```

```
    </div>
</body>
</html>
```

代码清单2-11添加了一个拥有asp-action属性的元素，asp-action属性是标签助手，它会在渲染视图时执行Razor指令。asp-action属性是将href属性添加到包含指向操作方法的URL的a元素中的指令。第24～26章将详述标签助手的工作方式，asp-action属性是最简单的一种标签助手，它告诉Razor插入一个URL，这个操作方法由正在渲染当前视图的同一控制器定义。你可以通过启动项目看到标签助手创建的链接，如图2-15所示。

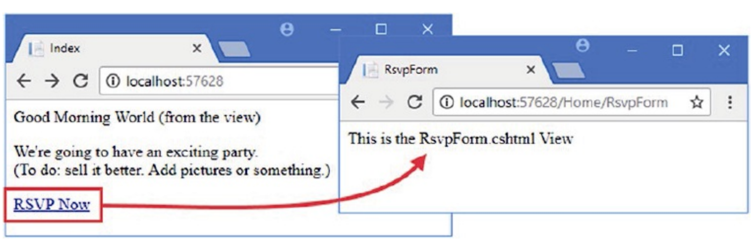


图2-15　链接两个操作方法

启动应用程序，在浏览器中移动光标到RSVP

Now链接上，你将看到链接指向以下URL（允许Visual Studio给你的项目分配不同的端口号）：

```
http://localhost:57628/Home/RsvpForm
```

这里有一个重要的原则，你应该使用MVC提供的特性（feature）来生成URL，而不是将它们硬编码到你的视图中。当使用标签助手为a元素创建href属性时，它会检查应用程序的配置以确定URL。这让应用程序的配置可以支持不同的URL格式，而不需要再去更新任何视图。第15章将解释原理。

2.5.5 建立表单

既然已经创建了强类型视图，并且可以从Index视图中访问它，接下来就构建RsvpForm.cshtml文件的内容并转换成用于编辑GuestResponse对象的HTML表单，如代码清单2-12

所示。

代码清单2-12　在Views/Home文件夹下的RsvpForm.cshtml文件中创建一个表单视图

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" /></p>
```

```
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</opti
on>
                <option value="true">Yes, I'll be there
</option>
                <option value="false">No, I can't come<
/option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

这里为GuestResponse模型类的每个属性都定义了一个label元素和一个input元素（或为WillAttend属性定义了一个select元素）。标签助手的另一个功能就是把每个元素都使用asp-for属性与模型属性关联起来。标签助手属性可通过配置将元素绑定到模型对象。下面是使用标签助手生成HTML并将其发送到浏览器的示例：

```
<p>
    <label for="Name">Your name:</label>
    <input type="text" id="Name" name="Name" value="">
```

```
</p>
```

这里使用label元素的asp-for属性设置了for属性的值，使用input元素的asp-for属性设置了id和name元素的值。这在目前看起来并不是特别有用，但是你以后会发现将元素与模型属性相关联会提供额外的优势。

更直接的是应用于form元素的asp-action属性，它通过应用程序的URL路由配置将action属性设置为一个URL，该URL将针对特定的操作方法，如下所示：

```
<form method="post" action="/Home/RsvpForm">
```

与应用于元素的helper属性一样，这种方法的好处是可以更改应用程序使用的系统URL，而由标签助手生成的内容也将自适应这些更改。

你可以通过运行应用程序并单击RSVP Now链

接（见图2-15）来查看表单，如图2-16所示。

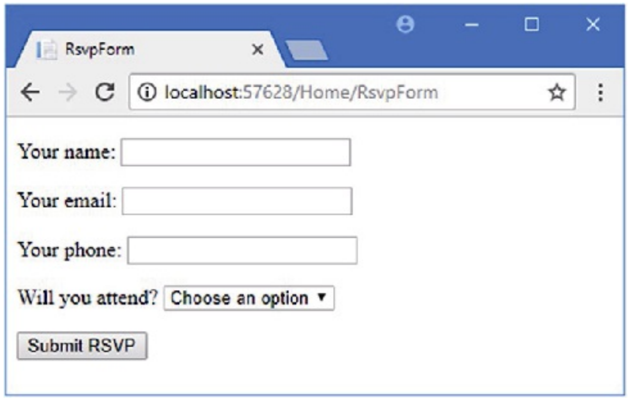


图2-16　查看表单

2.5.6　接收表单数据

当把表单发布到服务器时，我们还没有告诉MVC我们要做什么。现在单击Submit RSVP按钮只会清除我们已输入表单的所有值。这是因为在Home控制器中表单仅仅返回了同一个RsvpForm的操作方法，它只告诉MVC再次渲染视图（所以表单内容没有保存而被刷新掉）。

为了接收和处理提交的表单数据，这里将使用

一个核心的控制器特性。这里将添加第二个RsvpForm 操作方法来创建以下内容：

- 响应HTTP GET请求的方法。GET请求是当用户每次单击链接时浏览器发出的请求。在该例中，这个操作方法将负责在第一次访问/Home/RsvpForm时显示初始的空白表单。
- 响应HTTP POST请求的方法。默认情况下，使用Html.BeginForm()渲染的表单在被浏览器提交时，将发出POST请求。在该例中，这个操作方法将负责接收提交的数据并决定如何处理它们。

在独立的C#方法中处理GET和POST请求有助于保持控制器代码整洁，因为这两种方法有不同的职责。虽然这两个操作方法都调用同一个URL，但是MVC会基于使用的是GET还是POST请求来采取不同的方法处理请求。代码清单2-13显示了对HomeController类所做的更改。

代码清单2-13　在HomeController.cs文件中添加一个操作方法来支持POST请求

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public ViewResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestR
esponse) {
            // TODO: store response from guest
            return View();
        }
    }
}
```

这里已经将HttpGet特性添加到现有的

RsvpForm操作方法中，MVC知道该方法只可用于GET请求。然后，添加了一个重载版本的RsvpForm操作方法，该方法接收一个GuestResponse对象。如果将HttpPost特性应用于此方法，MVC会使用这个方法处理POST请求。随后几节将解释这些附加的操作是如何工作的。这里还导入了PartyInvites.Models命名空间，这样就可以引用GuestResponse模型类型，而不需要类名。

1．使用模型绑定

第一个重载的RsvpForm操作方法与前面的RsvpForm.cshtml文件渲染的是同一个视图，参见图2-16所示的表单。第二个重载的参数变得较有趣，考虑到操作方法是在响应HTTP POST请求时被调用的，而GuestResponse是一个C#类，二者是如何连接的呢?

答案是模型绑定（Model Binding），这是一个很有用的MVC特性，通过解析传入的数据，并使用HTTP请求中的键/值对填充领域模型类型的属性。

模型绑定是一种强大且可定制的特性，它消除了直接处理HTTP请求带来的烦琐工作，让你直接操作C#对象，而不是处理浏览器发送的单个数据值。GuestResponse对象将作为参数传递给操作方法，该对象会自动填充来自表单字段的数据。第26章将详细介绍模型绑定的细节，包括如何定制它。

这个应用程序的最后一个目标是提供总结页面，其中包含参加聚会的人员信息，这意味着需要跟踪收到的响应。通过创建对象的内存集合来实现这一点。虽然在实际应用中这可能不太好用，因为当应用程序停止或重新启动时，响应数据将丢失，但是这种方法将允许开发人员保持对MVC的持续关注，并创建一个很容易被重置为初始状态的应用程

序。

提　示

第8章将演示MVC的持久存储和数据访问，这是更贴合实际使用的一个示例。

添加一个文件到项目的Models文件夹并右击，选择Add→Class，将文件命名为Repository.cs并使用它来定义类，如代码清单2-14所示。

代码清单2-14　Models文件夹下的Repository.cs文件的内容

```
using System.Collections.Generic;

namespace PartyInvites.Models {
```

```
    public static class Repository {
        private static List<GuestResponse> responses =
new List<GuestResponse>();

        public static IEnumerable<GuestResponse> Respon
ses {
            get {
                return responses;
            }
        }

        public static void AddResponse(GuestResponse re
sponse) {
            responses.Add(response);
        }
    }
}
```

Repository类及其成员是静态的，这将使开发人员能够轻松地从应用程序的不同位置存储和检索数据。MVC提供了一种更复杂的方法来定义通用的功能，称为依赖注入（dependency injection），这将在第18章中详述，但静态类是一种很好的方法，可以用于简单的程序。

2．存储响应

既然可以存储数据了，接下来就更新接收HTTP POST请求的操作方法，如代码清单2-15所示。

代码清单2-15　更新HomeController.cs文件中的操作方法

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public ViewResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestR
esponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
```

```
        }
    }
}
```

所要做的就是处理发送到请求中的表单数据，比如处理传递给操作方法的GuestResponse对象，将其作为参数传递给Repository.AddResponse方法，以存储响应。

为什么模型绑定不像**Web**表单绑定？

第1章解释了传统ASP.NET Web Forms的一个缺点是将HTTP和HTML的细节对开发人员隐藏。你可能想知道，在代码清单2-15中用于从HTTP POST请求创建GuestResponse对象的MVC模型绑定是否也在做同样的事情。

答案为不是。模型绑定去除了烦琐和易导致错

误的任务，使开发人员不需要检查HTTP请求和提取每一个数值，但（这是最重要的部分）如果开发人员想手动处理一个请求，仍然可以使用模型绑定，因为MVC提供了对所有请求数据的轻松访问方式。MVC没有对开发人员隐藏任何东西（所有数据和操作都是可见的），而且其中还有许多特性可以更简易地使用HTTP和HTML。当然，为了使用这些特性，你需要手动选择。

这似乎是一个细微的差别，但是随着你对MVC的了解，你将感受到这种开发体验与传统ASP.NET Web Forms是完全不同的，并且你总是可以知道如何处理应用程序接收的请求。

在RsvpForm操作方法中调用View方法，使得MVC渲染一个名为Thanks的视图，并将

GuestResponse对象传递给视图。要创建视图，可在Solution Explorer中右击View/Home文件夹并从弹出菜单中选择Add→New Item。在ASP.NET类别中选择MVC View Page模板，命名为Thanks.cshtml，单击Add按钮。Visual Studio将创建Views/Home/Thanks.cshtml文件并打开它进行编辑。更改该文件的内容，如代码清单2-16所示。

代码清单2-16　Views/Home文件夹下的Thanks.cshtml文件的内容

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Thanks</title>
</head>
<body>
```

```
    <p>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks
 are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but
 thanks for letting us know.
        }
    </p>
    <p>Click <a asp-action="ListResponses">here</a> to
see who is coming.</p>
</body>
</html>
```

Thanks.cshtml视图文件基于在RsvpForm操作方法中传递给View方法的GuestResponse属性的值并使用Razor来显示内容。Razor的@model表达式指定了强类型视图的领域模型类型。

使用Model.PropertyName来访问领域对象中的属性值。举一个示例，要获取Name属性的值，可调用Model.Name。不用担心Razor的语法是没有意义的——第5章将更详细地解释它。

现在Thanks视图已经创建了，这是一个使用MVC处理表单的基本示例。在Visual Studio中，从Debug菜单中选择Start Debugging以启动应用程序，单击RSVP Now链接，向表单添加一些数据，然后单击Submit RSVP按钮。你将看到图2-17所示的结果（当名字不是Joe或者你不能参加聚会时，会显示不同的感谢信息）。

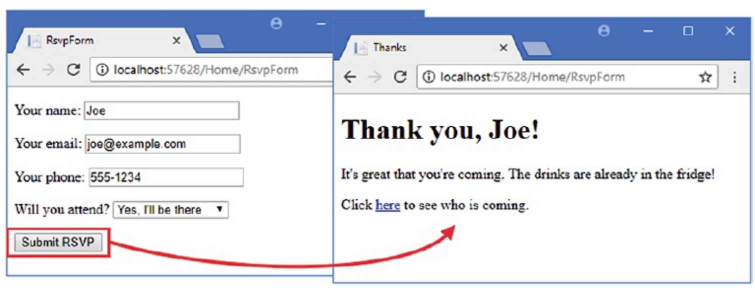


图2-17　Thanks视图

2.5.7　显示响应

在Thanks.cshtml文件的最后，添加一个a元素来创建一个链接以显示即将参加聚会的人的列表。下面使用asp-action tag helper属性创建一个URL，

该URL指向一个名为ListResponse的操作方法，如下所示：

```
...
<p>Click <a asp-action="ListResponses">here</a> to see
who is coming.</p>
...
```

如果将鼠标指针悬停在浏览器的链接上，你将看到它指向/Home/ListResponses这个URL。这与Home控制器中的任何操作方法都不对应，如果单击链接，就会看到一个空页面。打开浏览器的开发者工具并查看服务器发送的响应，将显示404 Not Found（Chrome浏览器有点奇怪，它不向用户显示错误消息，但是第14章会解释如何生成有意义的错误消息）。

这里将通过创建使URL对应的目标为Home控制器的操作方法来解决这个问题，如代码清单2-17所示。

代码清单2-17　在Controllers文件夹下的HomeController.cs文件中添加一个操作方法

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public ViewResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestR
esponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
        }

        public ViewResult ListResponses() {
            return View(Repository.Responses.Where(r =>
 r.WillAttend == true));
```

```
            }
        }
}
```

新的操作方法名为ListResponse，它使用Repository.Responses属性作为参数调用View方法。这就是操作方法为强类型视图提供数据的方式。对GuestResponse对象的集合使用LINQ进行过滤，使得只有将要参加聚会的响应被使用。

ListResponse这个操作方法没有指定应该用于显示GuestResponse对象集合的视图名称，这意味着将使用默认的命名约定，MVC将在Views/Home和Views/Shared文件夹中查找名为ListResponses.cshtml的视图。可在Solution Explorer中右击View/ Home文件夹，并从弹出菜单中选择Add→New Item以创建视图。在ASP.NET类别中选择MVC View Page模板，命名为ListResponses.cshtml并单击Add按钮。编辑新视图

的内容，如代码清单2-18所示。

代码清单2-18　Views/Home文件夹下的ListResponses.cshtml文件的内容：显示邀请函

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{

    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Responses</title>
</head>
<body>
    <h2>Here is the list of people attending the party<
/h2>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email</th>
                <th>Phone</th>
            </tr>
        </thead>
        <tbody>
```

```
            @foreach (PartyInvites.Models.GuestResponse
 r in Model) {
                <tr>
                    <td>@r.Name</td>
                    <td>@r.Email</td>
                    <td>@r.Phone</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

Razor视图文件有.cshtml文件扩展名，因为它们是C#代码和HTML元素的混合体。你可以在代码清单2-18中看到这一点，这里使用foreach循环来处理每个GuestResponse对象，它在操作方法中使用View方法传递给视图。与原生的C#中的foreach循环不同，Razor的foreach循环的主体包含添加到响应中的HTML元素，这些元素将被发送回浏览器。在这个视图中，每个GuestResponse对象都会生成一个tr元素，tr元素中又会包含一个td元素，td元素中是对象属性的值。

要查看工作中的列表，可以通过从Debug菜单中选择Start Debugging来运行应用程序，提交一些表单数据，然后单击链接查看响应列表。你将看到自应用程序启动以来输入的所有数据，如图2-18所示。视图呈现数据的方式有些简陋，但是现在已经足够完成我们的功能了，本章后面将介绍应用程序的其他样式。

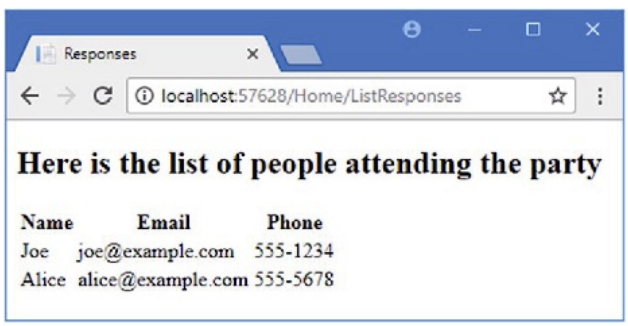


图2-18　输入的所有数据

2.5.8　添加验证

现在该向应用程序添加数据验证功能了。如果程序缺乏验证，用户就可以输入无意义的数据，甚至提交一个空表单。在MVC应用程序中，验证通常

应用于领域模型而不是用户界面。这意味着你只在一个地方定义了验证，但能够在模型类使用的应用程序的任何地方生效。MVC支持声明式验证规则（Declarative Validation Rule），这是通过System.ComponentModel.DataAnnotations命名空间中的Attribute进行定义的。这意味着验证约束是使用标准C#注解属性来表示的。代码清单2-19展示了如何将这些属性应用到GuestResponse模型类。

代码清单2-19　在Models文件夹下的GuestResponse.cs文件中添加验证以应用相关属性

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

    public class GuestResponse {

        [Required(ErrorMessage = "Please enter your nam
e")]
        public string Name { get; set; }
        [Required(ErrorMessage = "Please enter your ema
il address")]
        [RegularExpression(".+\\@.+\\..+",
            ErrorMessage = "Please enter a valid email
```

```
address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your pho
ne number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whethe
r you'll attend")]
        public bool? WillAttend { get; set; }
    }
}
```

MVC会自动检测属性并在模型绑定过程中使用它们来验证数据。由于已经导入包含验证属性的命名空间，因此不再需要限定它们的名称就可以引用它们。

提　示

正如前面提到的，WillAttend属性使用可为null

的bool型。这样做是为了能够应用Required注解属性。如果使用常规的bool型，那么通过模型绑定接收到的值只有true或false，你将无法判断用户是否选择了值。可空的bool型有三个可能的值——true、false和null。如果用户没有选择值，浏览器将发送null，这让Required注解属性能够报告验证错误。这是MVC将C#特性与HTML和HTTP混合在一起的一个很好的示例。

你可以在控制器类中使用ModelState.IsValid属性来检查是否存在验证问题。代码清单2-20显示了如何在Home控制器类里启用POST的RsvpForm操作方法中检查表单验证错误。

代码清单2-20　在Controllers文件夹下的HomeController.cs中检查表单验证错误

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public ViewResult RsvpForm() {
            return View();
        }
        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestR
esponse) {
            if (ModelState.IsValid) {
                Repository.AddResponse(guestResponse);
                return View("Thanks", guestResponse);
            } else {
                // there is a validation error
                return View();
            }
        }

        public ViewResult ListResponses() {
            return View(Repository.Responses.Where(r =>
 r.WillAttend == true));
```

```
        }
    }
}
```

Controller基类提供了一个名为ModelState的属性，该属性提供关于将HTTP请求数据转换成C#对象的信息。如果ModelState.IsValue属性返回true，那么可以知道MVC能够满足通过GuestResponse类的属性指定的验证约束。因此验证通过，将展现Thanks视图。

如果ModelState.IsValue属性返回false，那么可以知道发生验证错误了。ModelState属性返回的对象提供了每个问题的详细信息，但开发人员并不需要知道细节，因为通过一个特别有用的特性，无须任何参数就能调用View方法，并自动地请求用户解决任何问题。

当MVC渲染视图时，Razor可以访问与请求相关的所有验证错误的细节，而标签助手可以访问细

节并向用户显示验证错误。代码清单2-21显示了如何为RsvpForm视图添加验证用的标签助手属性。

代码清单2-21　在位于Views/Home文件夹下的RsvpForm.cshtml文件中添加验证总结

```
@model PartyInvites.Models.GuestResponse

@{

    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <div asp-validation-summary="All"></div>
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
```

```
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" /></p>
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</opti
on>
                <option value="true">Yes, I'll be there
</option>
                <option value="false">No, I can't come<
/option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

将asp-validation-summary属性应用于div元素，并在渲染视图时显示验证错误列表。asp-validation-summary属性的值来自ValidationSummary枚举，这个枚举指定了验证总结中将包含哪些类型的验证错误。这里指定包含所有类型的验证错误，这是大多数应用程序开始编写时的常用方式，第27章将描述其他的取值并解释它们的工作原理。

要查看验证总结是如何工作的，运行应用程序，只填写Name字段，然后不输入任何其他数据就提交表单。你将看到的验证错误如图2-19所示。

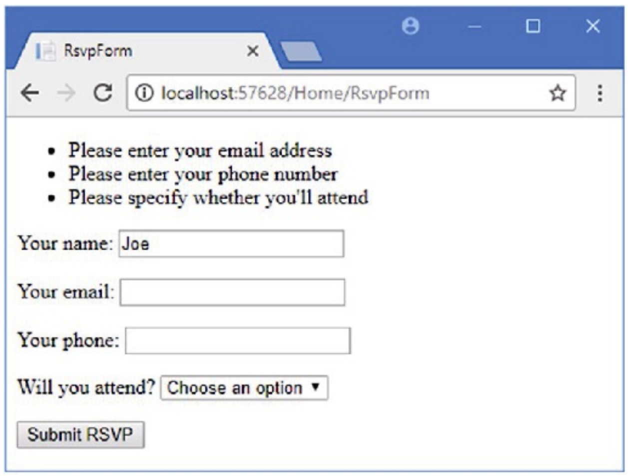


图2-19　验证错误

RsvpForm这个操作方法将不会渲染Thanks视图，直到满足应用到GuestResponse类的所有验证约束为止。注意，当Razor使用验证总结显示视图时，输入Name字段中的数据（合法数据）将被保留并再次显示。这是模型绑定的另一个好处——简化了表单数据的工作。

注　意

使用过ASP.NET Web Forms的读者知道服务器控件（server control）的概念，通过将序列化的值保存到一个名为__VIEWSTATE的隐藏字段中，服务器控件可保持状态。MVC模型绑定与服务器控件、回发或视图状态等Web表单概念无关。MVC不会将隐藏的__VIEWSTATE字段注入所渲染的HTML页面。相反，它通过设置输入元素的value属性来包含数据。

高亮显示无效字段

标签助手属性与模型属性通过元素关联了一个方便的特性，可以用来和模型绑定结合。当模型属性验证失败时，标签助手属性将生成略有不同的HTML。这里是在没有验证错误时为“手机号”字段生成的HTML：

```
<input type="text" data-val="true" data-val-required="P
lease enter your phone number"
id="Phone" name="Phone" value="">
```

相比之下，以下是在用户提交表单而不将任何数据输入文本字段后（这是验证错误，因为将Required验证属性应用到了GuestResponse类的Phone属性）生成的HTML：

```
<input type="text" class="input-validation-error" data-
val="true"
    data-val-required="Please enter your phone number"
id="Phone"
    name="Phone" value="">
```

这里用粗体高亮显示了不同之处——标签助手

属性将输入元素添加到一个名为input-validation-error的类中。我们可以利用这个特性创建一个样式表，其中包含这个类的一些CSS样式，以及其他不同的HTML标签助手属性以丰富显示效果。

MVC项目中的约定是将交付给客户的静态内容（如CSS样式表等）放置到wwwroot文件夹中，根据内容的类型进行组织排序，所以CSS样式表被放入wwwroot/css文件夹，JavaScript文件则被放入wwwroot/js文件夹，等等。

要创建一个CSS样式表，可以右击wwwroot/css文件夹，在Visual Studio的Solution Explorer中，选择Add→New Item，导航到客户端部分，并从模板列表中选择Style Sheet，如图2-20所示。

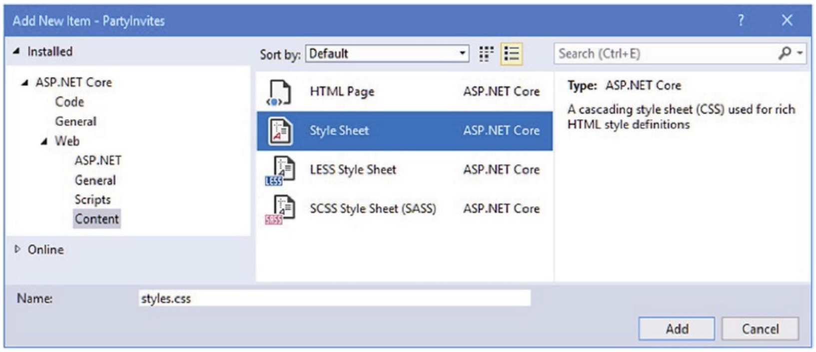


图2-20　创建一个CSS样式表



提　示

当使用Web Application（Model-View-Controller）模板创建项目时，Visual Studio会在wwwroot/css文件夹中创建site.css文件。可以忽略这个文件，你在本章中不会用到它。

将文件的名称设置为styles.css，单击Add按钮创建CSS样式表，编辑这个文件，如代码清单2-22所示。

代码清单2-22　wwwroot/css文件夹下的styles.css文件的内容

```
.field-validation-error    {color: #f00;}
.field-validation-valid    { display: none;}
.input-validation-error    { border: 1px solid #f00; ba
ckground-color: #fee; }
.validation-summary-errors { font-weight: bold; color:
#f00;}
.validation-summary-valid  { display: none;}
To apply this stylesheet, I have added a link element t
o the head section of the RsvpForm
view, as shown in Listing 2-23.
```

为了应用这个样式表，需要向RsvpForm视图的头部添加一个link元素，如代码清单2-23所示。

代码清单2-23　在Views/Home文件夹下的RsvpForm.cshtml文件中应用样式表

```
...
<head>
    <meta name="viewport" content="width=device-width"
```

```
/>
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
</head>
...
```

link元素使用href属性指定样式表的位置。注意，URL中省略了wwwroot文件夹。ASP.NET的默认配置包括对服务静态内容（例如图像、CSS样式表和JavaScript文件）的支持，并且自动将请求映射到wwwroot文件夹。第14章将详述ASP.NET和MVC配置过程。

提　示

在处理样式表时，有一个特殊的标签助手，如果有许多文件可以管理的话，使用这个标签助手将

十分方便。详见第25章。

随着样式表的应用，当提交数据时会突出显示验证错误，如图2-21所示。

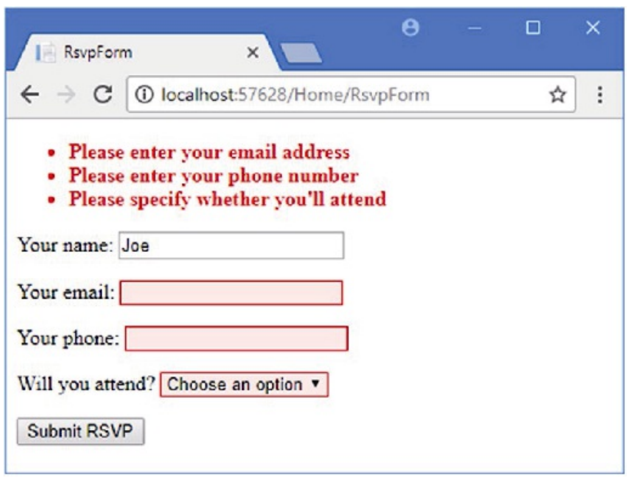


图2-21　自动突出显示的验证错误

2.5.9　设置内容样式

至此，应用程序的所有功能目标都已完成，但是应用程序的整体外观很差。当你使用Web Application（Model-View-Controller）模板创建项目

时，Visual Studio已经内置安装了一些常见的开源库。虽然作者不喜欢使用模板项目，但作者确实喜欢微软选择的库。其中一个叫作Bootstrap，它是一个很好的CSS框架，最初是由Twitter开发的，目前已经成为一个主要的开源项目，并且已经成为Web应用程序开发的主流框架之一。

注　意

在撰写本书时，Bootstrap 3是当前最新版本，并且第4代版本已在开发。微软可能会选择在Visual Studio的后期版本中更新Web Application（Model-View-Controller）模板所使用的Bootstrap版本，这可能会导致内容以不同的方式显示。这对于本书的

其他章节来说并不是问题，因为本书将展示如何显式地指定某个库的版本，以便得到预期的结果。

1．设置欢迎视图

Bootstrap的基本特性是通过运用于HTML元素的类起作用的，这些类与定义在wwwroot/lib/bootstrap文件夹中的CSS选择器相关联。Bootstrap的相关细节可以从getbootstrap网站获取。你也可以通过代码清单2-24中的MyView.cshtml视图文件看到如何将一些基本样式应用到视图文件。

代码清单2-24　添加Bootstrap到Views/Home文件夹下的MyView.cshtml文件中

```
@{
    Layout = null;
}
```

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body>
    <div class="text-center">
        <h3>We're going to have an exciting party!</h3>
        <h4>And you are invited</h4>
        <a class="btn btn-primary" asp-action="RsvpForm
">RSVP Now</a>
    </div>
</body>
</html>
```

这里添加了link元素，它的href属性可以用于从wwwroot/lib/bootstrap/dist/css文件夹加载bootstrap.css文件。通常约定将第三方的CSS和JavaScript包安装到wwwroot/lib文件夹中，第6章将详述用于管理这些包的工具。

导入Bootstrap样式表后，需要对元素进行样式

化。这只是一个简单的示例，所以只使用少量的Bootstrap CSS类——text-center、btn以及btn-primary。

text-center类可以使元素及子元素的内容居中。btn类用于修饰button、input或其他作为按钮使用的元素的样式，而btn-primary类用于指定按钮的颜色范围。你可以通过运行应用程序看到效果，如图2-22所示。

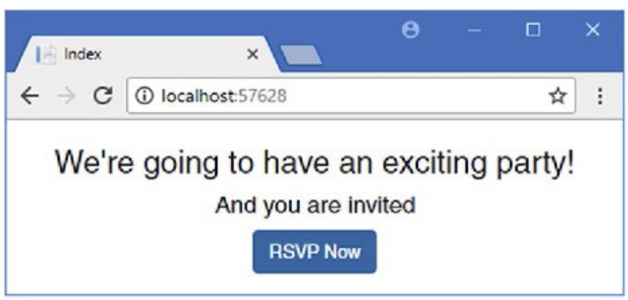


图2-22　样式化视图的效果

2．设置RsvpForm（电子回复）表单视图

Bootstrap定义了可以用于样式表的类。这里不

打算详细介绍，但是会展示如何将这些类应用到代码清单2-25中。

代码清单2-25　在Views/Home文件夹下的RsvpForm.cshtml文件中添加Bootstrap

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body>
    <div class="panel panel-success">
        <div class="panel-heading text-center"><h4>RSVP
</h4></div>
        <div class="panel-body">
            <form class="p-a-1" asp-action="RsvpForm" m
ethod="post">
                <div asp-validation-summary="All"></div>
```

```
                <div class="form-group">
                    <label asp-for="Name">Your name:</l
abel>
                    <input class="form-control" asp-for
="Name" />
                </div>
                <div class="form-group">
                    <label asp-for="Email">Your email:<
/label>
                    <input class="form-control" asp-for
="Email" />
                </div>
                <div class="form-group">
                    <label asp-for="Phone">Your phone:<
/label>
                    <input class="form-control" asp-for
="Phone" />
                </div>
                <div class="form-group">
                    <label>Will you attend?</label>
                    <select class="form-control" asp-fo
r="WillAttend">
                        <option value="">Choose an opti
on</option>
                        <option value="true">Yes, I'll 
be there</option>
                        <option value="false">No, I can
't come</option>
                    </select>
                </div>
                <div class="text-center">
                    <button class="btn btn-primary" typ
e="submit">
                        Submit RSVP
                    </button>
```

```
                </div>
            </form>
        </div>
    </div>
</body>
</html>
```

这个示例中的Bootstrap类创建了一个标题，只是为了提供一下结构布局。为了对表单进行样式化，使用了form-group类，该类用于对包含标签和相关输入或选择的元素进行样式化。你可以在图2-23中看到样式的效果。

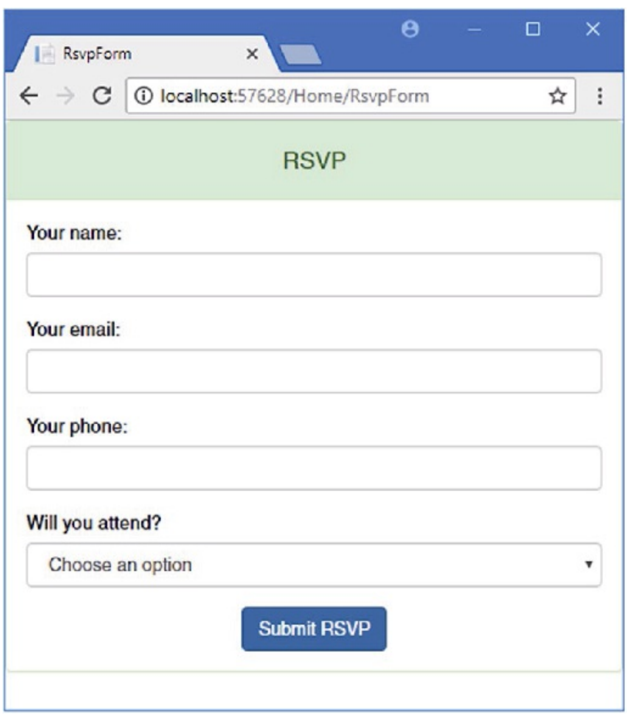


图2-23　样式化RsvpForm视图的效果

3．样式化Thanks视图

下一个要样式化的视图文件是Thanks.cshtml，你可以在代码清单2-26中看到具体是如何完成的，使用的CSS类与在其他视图中使用的相似。为了使应用程序更容易管理，尽可能避免重复代码和标记是一条很好的原则。MVC提供了几个特性来帮助减少重复，后面的章节将对此进行描述。这些特性包括Razor布局、分部视图和视图组件。

代码清单2-26　在Thanks.cshtml文件中使用Bootstrap

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
```

```
    <meta name="viewport" content="width=device-width"
/>
    <title>Thanks</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body class="text-center">
    <p>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks
 are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but
 thanks for letting us know.
        }
    </p>
    Click <a class="nav-link" asp-action="ListResponses
">here</a>
    to see who is coming.
</body>
</html>
```

图2-24显示了样式化Thanks视图的效果。

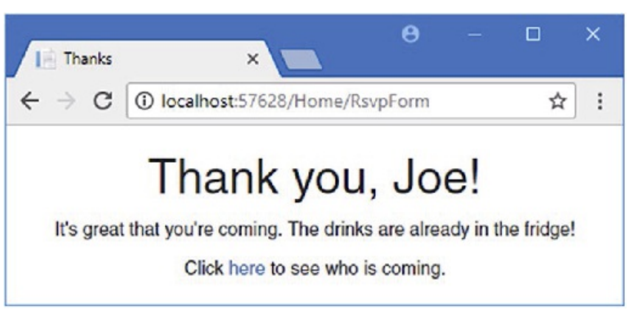


图2-24　样式化Thanks视图的效果

4．样式化列表视图

最后一个需要样式化的视图是ListResponses，它是一个用于展示参加派对人员信息的列表。在样式化内容时遵循与所有Bootstrap样式相同的基本操作，如代码清单2-27所示。

代码清单2-27　在Views/Home文件夹下的ListResponses.cshtml文件中添加Bootstrap

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
    <title>Responses</title>
</head>
<body>
    <div class="panel-body">
```

```
        <h2>Here is the list of people attending the pa
rty</h2>
        <table class="table table-sm table-striped tabl
e-bordered">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th>Phone</th>
                </tr>
            </thead>
            <tbody>
                @foreach (PartyInvites.Models.GuestResp
onse r in Model) {
                    <tr>
                        <td>@r.Name</td>
                        <td>@r.Email</td>
                        <td>@r.Phone</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

图2-25展示了样式化ListResponses视图的效果。将这些样式添加到视图中，即可完成这个应用程序。应用程序现在已满足所有的开发目标，并且外观上也有了很大的改进。

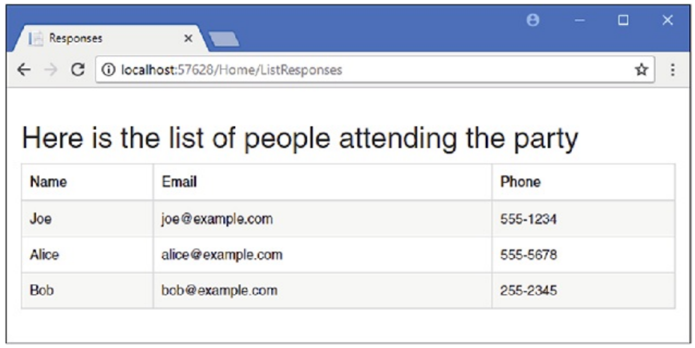


图2-25　样式化ListResponses视图的效果

2.6　小结

在本章中，我们创建了一个新的 MVC 项目，并使用它构造了一个简单的数据录入程序，你看到了ASP.NET Core MVC的大体架构和方法。本章跳过了一些关键特性（包括Razor语法、路由和测试），但是后面的章节将深入讨论这些主题。下一章将描述MVC的设计模式，这是使用ASP.NET Core MVC实现有效开发的基础。

第3章　MVC模式、项目与约定

在深入探讨ASP.NET Core MVC的核心内容之前，最好熟悉一下MVC的设计模式、隐藏在后面的核心思想，以及将MVC项目转换为ASP.NET Core MVC项目的方法。你可能已经对本章将要讨论的一些想法和约定有所了解，特别是如果之前做过ASP.NET或C#开发工作的话。如果没有，建议仔细研读一下本章，以加深对MVC底层知识的理解，从而帮助你更好地投入到本书接下来的框架特性学习中。

3.1　MVC简史

MVC（Model-View-Controller，模型-视图-控制器）在20世纪70年代后期就已经出现，产生于

Xerox PARC（施乐公司的帕洛阿尔托研究中心）的Smalltalk项目，当时被构思为早期GUI应用程序的一种组织方式。最初的MVC模式的某些细节依赖于Smalltalk特有的概念，如屏幕和工具等，但是其更广泛的概念仍然适用于现在的应用程序，而且特别适用于Web应用程序。

3.2　MVC模式

从高级术语角度看，MVC模式意味着一个MVC应用程序将被分成至少3部分。

- 模型，它包含或代表用户使用的数据。
- 视图，它用来呈现模型中的一些用户界面。
- 控制器，它用来处理输入请求，执行模型操作，并选择渲染给用户的视图。

MVC架构的每一部分都是被明确界定和自包含的，我们称之为关注点分离。模型中操纵数据的

逻辑仅仅在模型内部，展示数据的逻辑仅仅在视图中，处理用户请求和输入的代码仅仅包含在控制器中，每一部分都有明确的分工。这样，无论应用程序有多大，在整个生命周期中它都会变得易于维护和扩充。

3.2.1　模型

模型中包含了用户使用的数据。有两种类型的模型：一种是视图模型，只代表从控制器传递到视图的数据；另一种是领域模型，包含业务域中的数据，以及创建、存储和操纵这些数据的操作、转换和规则，这些统称为模型逻辑。

模型是应用程序工作的定义域。例如，在银行应用程序中，模型代表了应用程序支持的任何事情，比如账户、总账、客户的信用额度以及模型中可以操控数据的操作，如存款和提款。模型也负责

维护整体的状态和数据的一致性，比如，确保所有的交易都被添加到分类账中并且客户没有从账户中取走超出账户余额的钱。

对于MVC的每个组件，接下来分别描述哪些内容应该包含在内，哪些内容不应该包含在内。在应用程序中使用MVC构建模型时，注意以下几点：

- 包含领域数据；
- 包含用户创建、管理和修改领域数据的逻辑；
- 提供一个可以公开模型数据和操作的干净的API。

以下做法是错误的：

- 暴露模型数据是怎样被获取或组织的（换句话说，数据存储机制的细节不能暴露给控制器和视图）；
- 包含基于用户交互改变模型的逻辑（因为那是控制器的工作）；

- 包含向用户显示数据的逻辑（因为那是视图的工作）。

保证模型与控制器和视图逻辑分离的好处是可以使逻辑的测试工作更加简单（第7章将会介绍单元测试）。不仅如此，这样做还可以使整个应用程序的优化和维护工作变得简单。

提　示

许多不熟悉MVC模式的开发者对数据模型中包含逻辑处理的事情感到困惑，请相信，MVC模式的目标是将数据和逻辑分离。有些人认为MVC模式的目标是将一个应用划分成3个功能区域，每个功能区域都包含逻辑和数据模块，其实这是误解。

MVC模式的目标并不是要从模型中消除逻辑，相反，而是确保模型中只包含用于创建和管理模型数据的逻辑。

3.2.2　控制器

控制器在MVC模型中是数据模型和视图之间联系的纽带。控制器定义了对数据模型进行操作的业务逻辑的动作，以及视图给用户提供视图数据的动作。

使用 MVC 模式构建控制器时应该包含基于用户交互更新模型的动作；不应该包含管理数据外观的逻辑（那是视图的工作），同时不应该包含管理数据持久化的逻辑（那是模型的工作）。

3.2.3　视图

视图包含了向用户展示数据或者从用户那里获取数据以便控制器使用的逻辑。视图应该包含需要展示给用户的逻辑和标注，不应该包含复杂的逻辑（这些最好放在控制器中）和创建、存储或操控域模型的逻辑。

视图可以包含逻辑，但是包含的逻辑应该比较简单或者很少使用。在视图中放置的任何事情，即使是最简单的方法调用或表达式，也会使整个应用变得难以测试和维护。

3.2.4　MVC的ASP.NET实现

顾名思义，ASP.NET Core MVC使MVC模式适用于ASP.NET和C#开发。在ASP.NET Core MVC中，控制器是C#类，通常派生于微软的AspNetCore.Mvc.Controller类。从控制器派生而来的类中的每个公有方法都是一种操作方法，与一个

URL相关联。当一个请求被发送给一个与操作方法相关联的URL时，便会执行操作方法中的语句，以执行领域模型上的一些操作，然后选择一个视图显示给客户端。图3-1显示了控制器、模型和视图之间的这种交互。

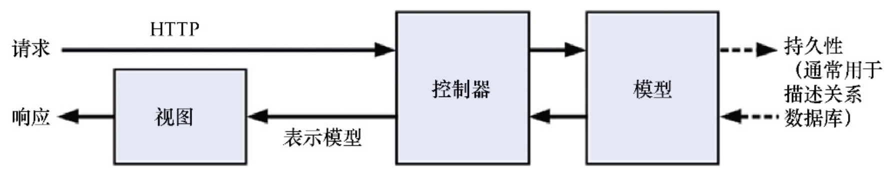


图3-1　MVC应用程序中的交互

ASP.NET Core MVC使用名为Razor的视图引擎组件来负责处理视图，以生成浏览器需要的结果。Razor视图是一些包含C#逻辑的HTML模板，用于处理模型数据以产生随着模型变化而生成的动态内容。第5章将会介绍Razor的工作原理。

ASP.NET Core MVC对领域模型的实现没有任何约束。你可以使用常规的C#对象创建模型，并且

可以使用.NET支持的任何数据库、对象关系映射（ORM）框架或者.NET支持的其他数据工具来实现持久化。

单页应用程序

Web应用程序开发历史倾向于将浏览器视为呈现HTML和响应鼠标单击的简单显示设备，这称为Web应用程序的往返样式。每次用户单击链接时，都会向ASP.NET Core MVC应用程序发送一个HTTP请求，在这种应用程序中，控制器选择一个由Razor呈现并发送回浏览器的视图，以便向用户显示一个新的HTML页面。所有的逻辑、数据和状态都存在于ASP.NET Core MVC服务器中，这简化了开发过程，意味着除了确保浏览器能够处理Razor视图中包含的HTML功能外，不必太关注浏览器。

相比之下，单页应用程序将浏览器合并到应用程序平台中。服务器负责管理应用程序的数据，而运行浏览器的JavaScript代码则请求数据，将其显示给用户，并响应用户交互。在单页应用程序中，模型、视图和控制器共享浏览器和服务器。

应用程序的ASP.NET Core MVC服务器部分提供对应用程序数据的访问，而不是将完整的HTML页面发送到浏览器，这些数据是由JavaScript框架（如Angular或React）查询和显示的。

单页应用程序可能比往返应用程序更具响应性，但它们的创建更复杂，需要同时具备C#和JavaScript技能才能进行有效的开发，难度不可低估。第20章将演示如何使用ASP.NET Core MVC来提供此类应用程序中的数据，但不会演示单页应用程序开发。作者最喜欢的JavaScript框架是Angular，如果要在ASP.NET Core MVC中使用

Angular，请参阅作者编写的另一本书——*Angular for ASP.NET Core MVC*。

3.3　MVC与其他模式的比较

当然，MVC并不是唯一的软件架构模式。还有很多其他的模式，并且其中的一些至少曾经是非常流行的。通过比较，你也可以了解关于MVC的更多知识。接下来的章节将简要介绍几个创建应用的不同方法，并将它们与MVC进行对比。一些与MVC大同小异，其他一些则完全不同。

并不是说MVC对所有场景都是最佳模式。作者主张选取最佳方法来解决手头的问题。你会看到，在某些场景下，有些其他模式与MVC的表现相当，甚至要优于MVC。在选择模式时，建议做出明

智的选择。

3.3.1 “智能UI”模式

最常用的设计模式之一称为“智能UI”模式。大多数程序员在其职业生涯中有过创建智能UI程序的经历。如果使用过Windows Forms或ASP.NET Web Forms，那你一定也有这样的经历。

要创建一个UI应用程序，通常，开发人员会通过把一组组件或控件拖放到设计界面或画布的方式来构建用户界面。控件可以创建用户的按压按钮事件、单击事件、鼠标移动事件以与用户交互。开发人员为控件的这些事件编写一系列事件处理代码。当特定组件的特定事件被触发时便会调用这些代码。这就创建了一种单片式的应用程序，如图3-2所示。处理用户界面与业务逻辑的代码是混合在一起的，它们没有做分离。定义输入数据、执行数据

查询或修改用户账号的代码片段都已按期望的事件顺序耦合在一起。

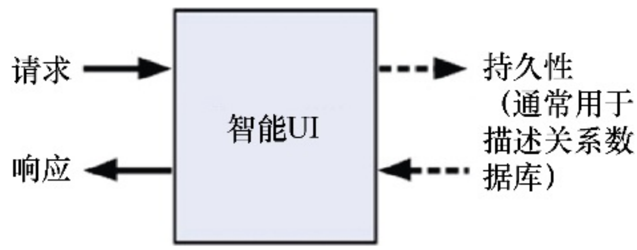


图3-2　智能UI模式

智能UI项目非常适合简单的项目，因为你很快就可得到一些很好的结果（你将会在第8章看到，MVC开发在交付结果之前需要做一些精心的准备工作和前期投入）。智能UI项目也适用于用户界面的原型开发。这些界面设计工具确实是很不错的。如果你正在与客户沟通并且正需要捕获客户对外观和接口流程的需求，那么智能UI系统可以帮助你以快速响应的方式生成和测试一些不同的方案。

智能UI模式最大的缺点就是难以维护和扩展。

将领域模型与带有用户界面代码的业务逻辑混搭在一起会导致工作的重复。换言之，为了支持新增加的组件，需要复制并粘贴相同的业务逻辑代码片段。查找所有重复的部分并修改错误是非常难的事情。要添加新的功能而不影响已有的功能是几乎不可能的，对智能UI应用程序进行测试也是一件难事。唯一的办法就是模拟用户的交互行为，这项工作不仅效果不理想，而且很难保证有一个全面的测试覆盖率。

在MVC中，智能UI模式通常是一种反模式（anti-pattern）。这种反模式的出现，至少在某些地方，是因为大多数人是在经历了长期的智能UI应用程序开发与维护导致的失控之后，才来到MVC世界以寻求另一种开发方案。

另外，拒绝难以控制的智能UI模式是错误的。并不是智能UI模式中的所有东西都那么不堪，其中

也有很多积极的方面。智能UI模式是非常易于开发的，并且开发起来速度比较快。组件和设计工具方面的开发商投入很大的精力，使得用户的开发经历尽量愉悦，即使几乎没有任何开发经验的人在几小时内也可以开发出具有专业外观和合理功能的应用。

智能UI应用最大的弱点是可维护性差，但不会在小型开发项目中出现。如果要为小客户开发一款简单的工具，那么智能UI应用是很好的选择，MVC应用程序的额外复杂性则根本无法保证。

3.3.2 理解模型-视图架构

智能UI应用中可能会出现维护问题的地方是业务逻辑，业务逻辑横跨多个应用以至于改变或增加功能都是十分艰难的。模型-视图（model-view）架构则为此做了改进，它将业务逻辑抽取出来形成单

独的域模型。这样做之后，数据、处理过程和规则都集中在应用程序中的同一个部件中，如图3-3所示。

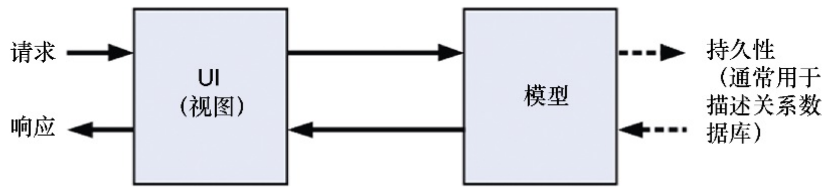


图3-3　模型-视图架构

模型-视图架构可以说是对智能UI模式的一种整体上的改进，比方说更加易于维护了，但同时也来带来两个问题。第一个问题，因为UI和域模型是紧密结合的，所以对其中任何一个进行单元测试都比较困难。第二个问题来自实践而不是模式的定义。模型通常会包含大量的数据访问代码，其实并不需要这样，但通常是这样的，这就意味着数据模型不仅仅包含业务数据、操作和规则。

3.3.3　经典的3层架构

为了解决“模型-视图”架构存在的问题，可使用3层模式（见图3-4）将持久性代码从域模型分离出来并放置在一个新的组件中，这个组件称作数据访问层（DAL）。

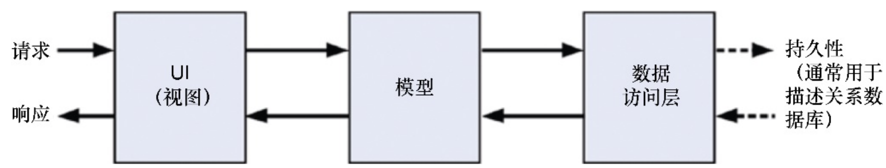


图3-4　3层模式

3层架构是应用程序中使用最广泛的模式。它对UI的实现方式没有限制，并且不需要很复杂的操作就可以实现很好的关注点分离（SoC）。另外，我们注意到，有了数据访问层（DAL），单元测试也会变得相对更容易一些。我们可以很容易地观察到经典的3层架构应用程序和MVC模式的相似之处。不同的是，当UI层直接耦合到GUI框架（例如Windows窗体或ASP.NET Web窗体）的click事件时，要自动地执行单元测试几乎是不可能的。由于

3层架构应用程序的UI部分可以很复杂，因此有很多代码无法严格测试。

最坏情况下，3层架构在UI层严重缺乏约束，也就是说，许多此类没有真正实现关注点分离（SoC）的应用程序，都被错误地认为是智能UI应用程序。随之而来的就是最坏结果——无法测试、难以维护且极其复杂的应用程序诞生了。

3.3.4 MVC的多样性

前面已经介绍了MVC应用程序的核心设计原则，尤其是ASP.NET Core MVC方面的内容，还解释了MVC模式与其他模式不同的方面以及为适应项目范围和目的对MVC所做的添加、调整和调节工作。下面简要介绍MVC最普遍的两种变化。了解这些变化对于使用ASP.NET Core MVC并没有太大的帮助，这里只是为了保证信息的完整，因为你将听

到许多在讨论软件模式时用到的术语。

1．MVP（Model-View-Presenter，模型-视图-呈现器）模式

MVP是MVC的一种变体，以便更容易地适应状态的变化，例如Windows窗体或ASP.NET Web窗体。为了得到最好的智能UI模式而不引起通常会伴随而来的错误，这值得尝试。

在这种模式中，呈现器具有与MVC控制器相同的职责，但它与状态化视图具有更直接的关系，根据用户的输入和动作，直接管理着UI组件中显示的数据。MVP模式有以下两种实现。

- 被动式视图（passive view）实现，在这种实现中，视图不包含逻辑。视图是UI控件的容器，由呈现器直接进行操纵。
- 监管控制器（supervising controller）实现，在这种实现中，视图可能要负责一些具有表现逻辑

的元素，如数据绑定，并被给予对领域模型数据源的引用。

这两种实现方式之间的差别涉及视图如何智能化。任何一种方式下，呈现器与GUI框架都是解耦的，这使得呈现器的逻辑更简单且更适合单元测试。

2．MVVM（Model-View-View Model，模型-视图-视图模型）模式

MVVM模式是MVC的最新变体，源于微软，并应用于WPF。在MVVM模式中，模型和视图具有与MVC相同的作用。不同的是MVVM中关于视图模型的概念，视图模型是用户界面的一种抽象表示，它既暴露了视图中待显示的数据，也暴露了能够通过UI进行的操作。与MVC控制器不同，MVVM视图模型没有视图（或者任何特定UI技术）的概念。MVVM视图使用WPF的绑定（binding）

特性，将视图控件暴露的属性（下拉菜单中的条目或按钮的单击效果）与视图模型暴露的属性双向地关联在一起。

提　示

MVC也使用术语视图模型（view model），但指的是一种简单的模型类，只用于从控制器向视图传递数据。而领域模型（domain model）则相反，指的是数据、操作以及规则的完整表示。

3.4　ASP.NET Core MVC项目

在创建一个新的ASP.NET MVC项目时，Visual Studio会为你提供项目中需要的一些初始内容选项。这样不仅可以为新手开发人员简化整个学习过程，并且可以应用一些节省时间的最佳实践来完成常见的功能和任务。作者对使用项目或代码模板的方法并不热衷。提供这种模板的初衷是好的，但是只拿来执行总觉得平庸。ASP.NET和MVC的特色之一就是在定制个性化的平台时有非常大的灵活性。Visual Studio创建和填充的这些项目、类和视图会让用户难以摆脱他人的样式，并且这些内容和配置太过普通，一般可用性也不大。可能微软也不知道用户需要什么样的应用程序，所以他们做的是涵盖所有的基础内容。但是这种笼统的方式反而让作者摒弃这些默认的内容。

建议从一个空的项目开始，添加需要的文件夹、文件和软件包，这样你不仅可以了解MVC的工

作方式，还可以完全操控应用程序中包含的所有内容。

当然，作者的偏好不应该完全代表你的开发经验，可能你会发现模板更好用，特别是对于ASP.NET新手以及那些还没有开发适合自己的开发样式的开发者来说。你可能还会觉得项目模板是十分有用的资源并且是想法的来源，但是在你完全了解它的工作方式之前，你应该谨慎地向应用程序中添加任何功能。

3.4.1 创建项目

当首次创建新的ASP.NET项目时，可以从以下3个基本的起点——Empty模板、Web API模板和Web Application（Model-View-Controller）模板之一开始，如图3-5所示。

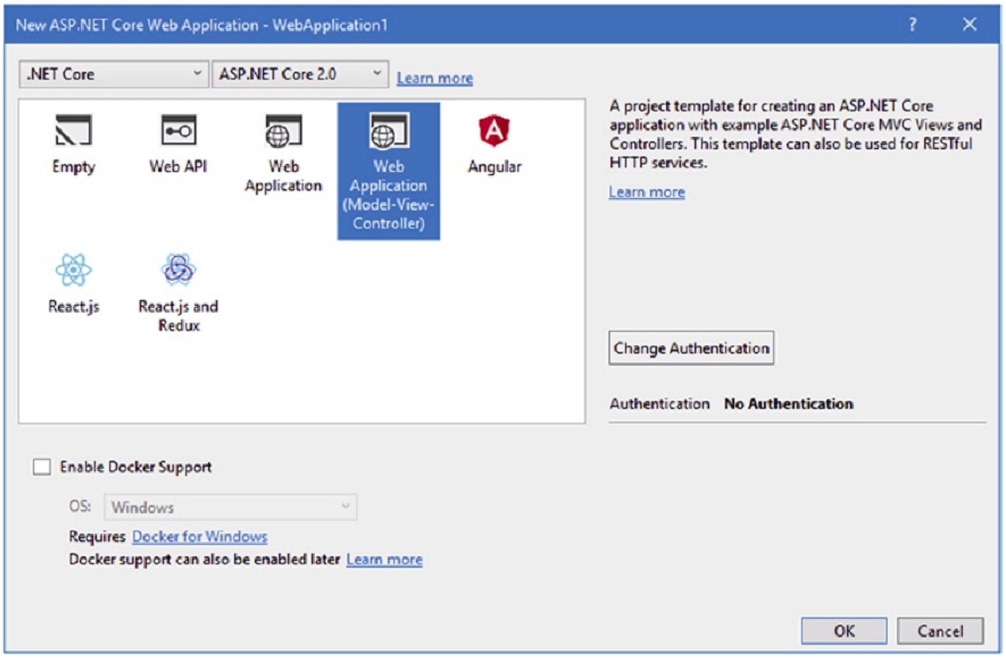


图3-5　ASP.NET项目模板

Empty模板包含ASP.NET的内核管道，但不包括MVC项目所需要的库或配置。Web API模板包含ASP.NET内核和MVC，其中的示例应用程序演示了如何接收和处理来自客户端的Ajax请求，第20章将对此进行描述。

Web Application（Model-View-Controller）模板包括ASP.NET Core和MVC，其中包含的示例应用程序演示了如何生成HTML内容。Web API和

Web Application（Model-View-Controller）模板可以提供不用的方案来为用户配置进行身份验证和授权访问功能。

其他模板提供了适合使用单页应用程序框架（Angular和React）与Razor页面的初始内容（允许代码和标记混合在一个文件中，合并控制器和视图的角色，并权衡MVC模型的一些优点以获得简单性）。

项目模板给人的印象可能是你需要遵循特定的路径来创建特定类型的 ASP.NET 应用程序，但情况并非如此。这些模板只是在完成相同功能时不同的起点。在使用任何模板创建的项目中都可以随时添加需要的功能。例如，处理Ajax请求，以及身份验证和授权功能都是从空的项目模板开始的。

因此，项目模板之间的真正区别是最初的库

集、配置文件、代码以及Visual Studio在创建项目时添加的内容。最简单的模板和最复杂的模板之间存在很多差异，图 3-6 展示了每种模板在创建新的项目之后的解决方案管理器（Solution Explorer）的样子。对于Web Application（Model-View-Controller）模板，因为单个列表太长了，所以必须将解决方案管理器放在不同的文件夹中。

使用Web Application（Model-View-Controller）模板添加到项目中的额外文件看起来令人望而生畏，但其中一些只是占位符或常见功能的示例实现。一些其他文件设置了MVC或配置了ASP.NET。其他的是客户端库，你将在程序生成HTML时添加这些库。现在的文件列表看起来可能比较长，但在你读完整本书后，你就会了解到每一个文件所要完成的功能。

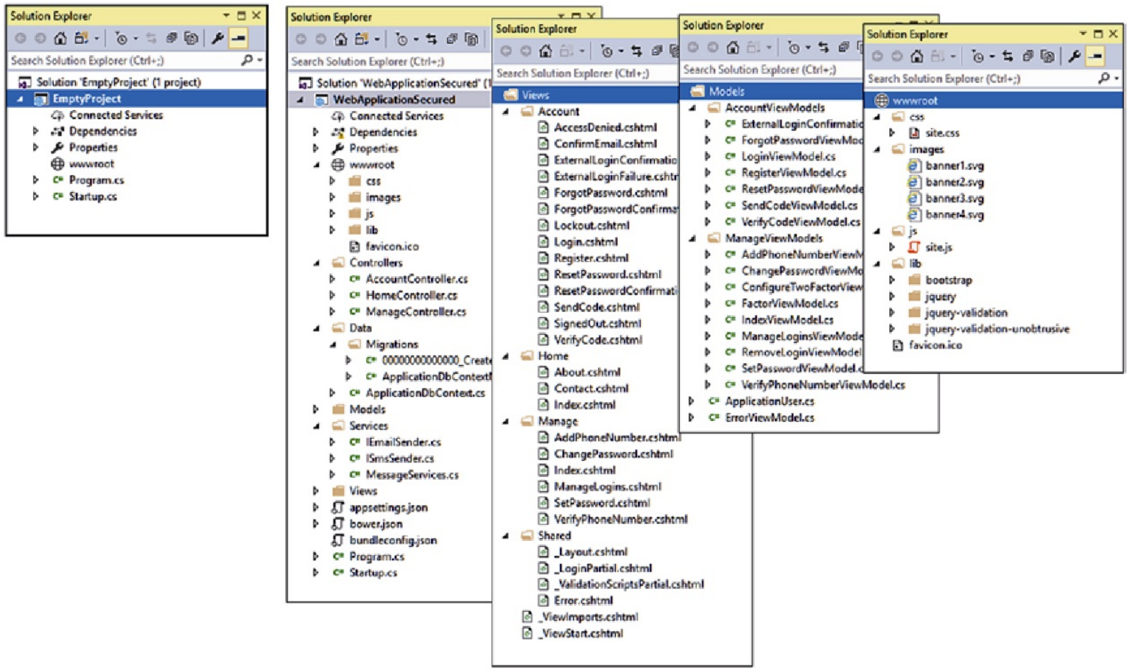


图3-6　Empty模板和Web Application（Model-View-Controller）模板添加到新项目中的默认内容

无论使用什么模板创建项目，都会发现一些公用的文件夹和文件是必然出现的。项目中的某些项具有特殊的功能，它们被硬编码到ASP.NET或MVC中，也可能包含在Visual Studio提供的支持工具中。其他的则取决于大多数ASP.NET或MVC项目使用的命名约定。表3-1描述了ASP.NET Core MVC项目中一些重要文件夹，其中一些文件夹默认

情况下不会出现，这会在稍后的章节中介绍。

表**3-1**　　重要的文件夹

文件夹	描述
/Areas	area是一种分解大型应用程序的方法
/Dependencies	提供对项目依赖的所有包的细节描述
/Components	定义视图组件类的地方，这些类用来显示自包含特征（例如购物车）
/Controllers	用来放置控制器类的地方，也可以将控制器类放在任何其他地方，它们都会被编译到同一个程序集中
/Data	这是定义数据库相关类的地方，但作者更习惯在Models文件中对它们进行定义
/Data/Migrations	这里存储了数据库架构的详细信息，以便存储应用程序的数据，在第8～11章中，作为SportsStore项目的一部分，使用了迁移
/Models	用来存放视图模型和域模型类，也可以在项目的任何地方或另一个单独的项目中定义模型类
/Views	保存视图和分部视图，通常与使用控制器命名的相关文件夹放在一起

/Views/Shared	这里包含非特定控制器的布局和视图
/Views/_ViewImports.cshtml	这个文件用于指定Razor视图中的命名空间，也可用来设置标签助手
/Views/_ViewStart.cshtml	这个文件用于指定Razor视图引擎的默认布局
/wwwroot	这是放置静态内容（如CSS文件和图像）的地方，同时也是Bower包管理器安装JavaScript和CSS软件包的地方

下面列出一些重要的文件。

- appsettings.json：包含可适应于不同环境（如开发、测试、生产环境）的配置信息。该文件常用于数据库服务器连接字符串和日志/调试配置。
- /bower.json：默认情况下，这个文件是隐藏的，其中包含了由Bower软件包管理器管理的文件列表。
- /<project>.csproj：用于指定项目的一些基本配置选项，包括使用的NuGet属性。
- /Program.cs：负责配置应用程序的宿主平台。

- /Startup.cs：用来配置应用程序。

3.4.2　关于MVC的约定

MVC项目中有两种约定。其中一种是关于如何构造项目的建议。例如，按常规将依赖的第三方JavaScript和CSS包放在wwwroot/lib文件夹中，其他MVC开发人员也希望在这里能找到这些包，软件包管理器也会默认将这些包安装在这里。当然，你也可以随意重命名或删除lib文件夹，或者把软件包放到其他地方。只要视图中的脚本和链接元素指向的位置正确，MVC就可以正常地运行应用程序。

另一种约定源于配置方面的原则，这也是Ruby on Rails如此受欢迎的原因之一。通过配置，不需要显式地配置控制器与视图之间的关联。例如，只需要遵循特定的文件命名规则就可以使程序正常执行。在此约定下，更改项目结构的灵活性就变得很

小了。

提　示

如果将标准的MVC组件替换为自己的实现，所有这些约定就可以改变了。本书将介绍不同的方法，以帮助我们解释MVC应用程序是如何工作的，这些是你在大多数项目中会碰到的约定。

1．关于控制器类的约定

控制器类的名称都是以Controller结尾的，例如ProductController、AdminController和

HomeController。当从项目的其他位置引用控制器时，如使用HTML辅助方法时，只需要指定名称的第一部分（如Product），MVC就会自动将Controller追加到名称中，并开始查找控制器类。

提　示

可以通过创建模型约定来改变这一点，详见第31章。

2．关于视图的约定

视图保存在/Views/Controllername文件夹中。

例如，与ProductController类关联的视图将保存在/Views/ Product文件夹中。

提　示

请注意，这里在Views文件夹中省略了控制器类的Controller部分，比如使用的是/Views/Product而不是Views/ProductController。

MVC希望操作方法的默认视图应以操作方法命名。例如，与列表（List）操作方法相关联的默认视图应名为List.cshtml。因此，对于ProductController类中的列表（List）操作方法，默

认视图应该是/Views/ Product/List.cshtml。当你在操作方法中返回调用View方法的结果时，将使用默认视图，如下所示：

```
...
return View();
...
```

要按名称指定不同的视图，如下所示：

```
...
return View("MyOtherView");
...
```

请注意，上面所列的视图没有包括文件扩展名或视图路径。在查找视图时，MVC首先会在以控制器命名的文件夹中查找，然后在/Views/Shared文件夹中查找。这意味着可以将多个控制器使用的视图放在/Views/Shared文件夹中，MVC可以找到它们。

3．关于布局的约定

布局的命名约定是在文件中加上下划线（_）字符，布局文件放在/Views/Shared文件夹中。默认情况下，所有视图都会使用/Views/_ViewStart.cshtml 文件。如果不想将默认版式应用于视图，可以更改_ViewStart.cshtml中的设置（或完全删除该文件）并在视图中指定其他版式，如下所示：

```
@{
    Layout = "~/_MyLayout.cshtml";
}
```

也可以对给定视图禁用任何布局，如下所示：

```
@{
    Layout = null;
}
```

3.5　小结

本章首先介绍了MVC架构模式，然后讨论了

领域模型的意义，并讲述了依赖注入。依赖注入能够消除组件耦合，以强制应用程序各部件之间严格分离。下一章将介绍C#语言特性，它们已广泛应用于MVC Web应用开发中。

第4章　C#基本特性

本章主要描述在Web应用开发中难理解或经常引起混淆的C#特性。因为这不是专门介绍C#的书，所以对于每个C#特性，本书只提供简单的示例，以便读者在后续的章节中能够更清楚地了解示例，并在自己的项目中充分利用这些特性。

表4-1列出了本章要解决的问题。

表**4-1**　本章要解决的问题

问题	解决方法	代码清单
避免访问空的属性引用	使用null条件控制运算符	代码清单4-5～代码清单4-8
简化C#属性	使用自动实现的属性	代码清单4-9～代码清单4-11
简化字符串组合	使用字符串插值	代码清单4-12

测试对象的类型或特征	使用模式匹配	代码清单4-13～代码清单4-16
一次性创建对象并设置其属性	使用对象或集合初始化器	代码清单4-17和代码清单4-18
为不能修改的类添加功能	使用扩展方法	代码清单4-19～代码清单4-26
简化委托和声明方法的使用	使用Lambda表达式	代码清单4-27～代码清单4-34
使用隐式类型	使用var关键字	代码清单4-35
创建对象而不定义其类型	使用匿名方法	代码清单4-36和代码清单4-37
简化异步方法的使用	使用async和await关键字	代码清单4-38～代码清单4-41
在不定义静态字符串的情况下获取类方法或属性的名称	使用nameof表达式	代码清单4-42和代码清单4-43

4.1　准备示例项目

在本章中，为了演示语言特性，使用ASP.NET

MVC Core Web Application（.NET Core）模板创建了一个新的Visual Studio项目，并命名为LanguageFeatures，如图4-1所示，取消勾选Add to Source Control复选框，单击OK按钮。

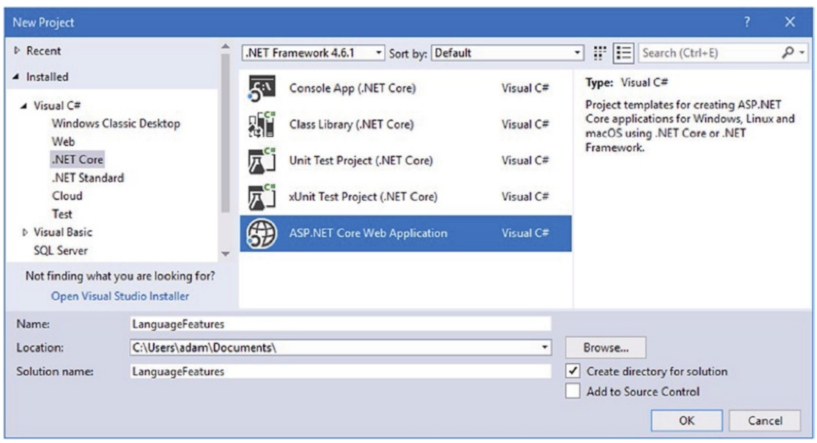


图4-1　选择项目类型

当需要呈现不同的项目配置时，可以选择Empty模板，如图4-2所示。在对话框顶部的列表中选择.NET Core和ASP.NET Core 2.0，确保将Authentication选项设置为No Authentication，并在单击OK按钮创建项目之前，取消选中Enable Docker Support复选框。

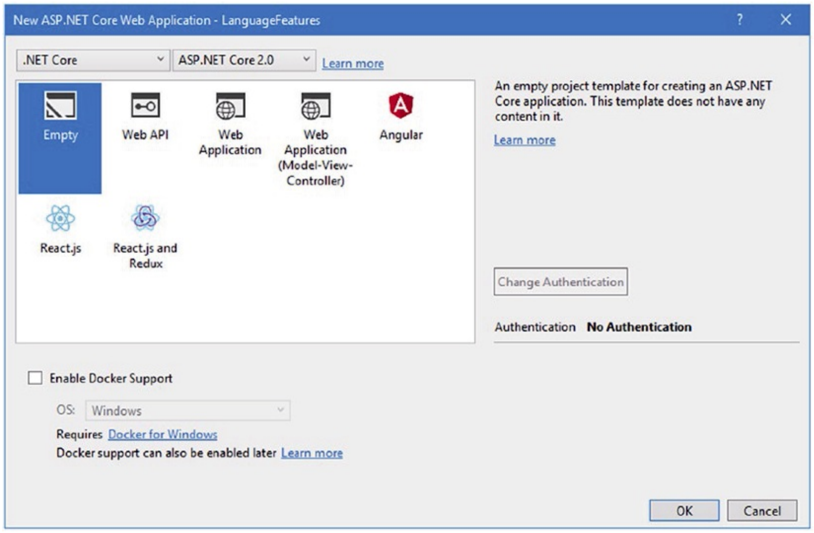


图4-2　选择项目模板

4.1.1　启用ASP.NET Core MVC

使用Empty项目模板创建的项目中包含最小的ASP.NET核心配置，并且没有提供任何MVC支持，这就意味着Web Application（Model-View-Controller）模板添加的占位内容尚不存在，要使控制器和视图起作用，还需要执行一些额外的操作步骤。本节虽然介绍了启用MVC设置所需要的步骤，但不会深入剖析每一个步骤的细节。

要启用MVC框架，请对Startup类进行代码清单4-1所示的更改。

代码清单4-1　在LanguageFeatures文件夹下的Startup.cs文件中启用MVC框架

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace LanguageFeatures {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            //app.Run(async (context) => {
```

```
            //    await context.Response.WriteAsync("He
llo World!");
            //});

            app.UseMvcWithDefaultRoute();
        }
    }
}
```

本书第14章将介绍怎样配置ASP.NET Core MVC应用程序，但代码清单4-1中额外添加的两个语句使得应用程序具有基本的MVC设置并使用默认的配置和约定。

4.1.2　创建MVC应用程序组件

MVC设置好了之后，我们便可以添加MVC应用程序组件来演示重要的C#语言特性。

1．创建模型

我们从创建一个简单的模型类开始，接下来就可以使用一些模型数据了。添加一个名为Models的

文件夹并在其中创建一个文件，命名为Product.cs以定义代码清单4-2中的类。

代码清单4-2　Models文件夹下的Product.cs文件的内容

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };

            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

Products类定义了Name和Price属性，还有一个名为GetProducts的静态方法用于返回Products数

组。这个数组中的一个元素已设置为null，用来展示一些有用的C#特性。

2．创建控制器和视图

对于本章中的示例，我们使用一个简单的控制器来演示不同的C#语言特性。创建Controllers文件夹并在其中添加HomeController.cs类文件，其中的内容如代码清单4-3所示。如果使用默认的MVC配置，MVC会默认把HTTP请求发送到Home控制器。

代码清单4-3　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View(new string[] { "C#", "Language"
, "Features" });
        }
    }
```

```
}
```

Index操作方法使得MVC呈现默认视图并且在HTML中传递一个字符串数组给客户端。为了创建相应的视图，添加Views/Home文件夹（首先创建Views文件夹，然后在其中创建Home子文件夹），再在其中创建Index.cshtml文件，这个文件的内容如代码清单4-4所示。

代码清单4-4　Views/Home文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Language Features</title>
</head>
<body>
    <ul>
        @foreach (string s in Model) {
            <li>@s</li>
        }
    </ul>
```

```
</body>
</html>
```

从Debug菜单中选择Start Debugging以运行示例应用程序，你会看到图4-3所示的输出。

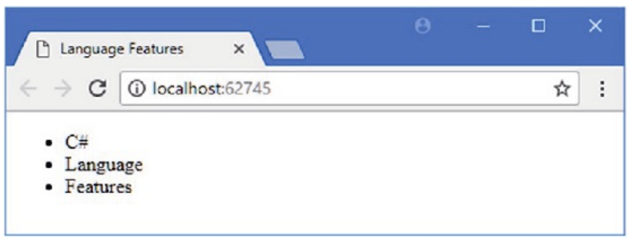


图4-3　运行示例应用程序

由于本章中所有示例的输出都是文本，因此浏览器中显示的信息可如下呈现：

```
C#
Language
Features
```

4.2　运用null条件运算符

null条件运算符可以使我们更优雅地判断null值。在MVC开发中，我们需要判断请求是否包含头

部或值，还需要判断模型中是否包含特殊的数据项，因此会有大量的null值需要检测。处理null值的传统方式是进行显式检测，但是当对象及其属性都需要检查时，这种方式就变得冗长且容易出错，null条件运算符使这一过程变得更加简洁明了，如代码清单4-5所示。

代码清单4-5　在Controllers文件夹下的HomeController.cs文件中检测null值

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()
) {
                string name = p?.Name;
                decimal? price = p?.Price;
                results.Add(string.Format("Name: {0}, P
rice: {1}", name, price));
```

```
            }

            return View(results);
        }
    }
}
```

Product类定义的GetProducts静态方法会返回一个对象数组。在控制器的Index操作方法中对这些值进行检查以获得Name和Price值的列表。问题是返回的对象以及对象中属性的值都有可能为null。也就是说，不能在for循环中仅仅通过p.Name或p.Price来引用值，否则会导致NullReferenceException。为了避免这一点，可使用null条件运算符，如下所示：

```
...
string name = p?.Name;
decimal? price = p?.Price;
...
```

null条件运算符是一个问号（符号？）。如果p为null，name也将设置为null；如果p不为null，

name将设置为Person.Name属性的值。同样，Price属性也是如此。请注意，在使用null条件运算符时，指派的变量必须能够为null，这就是为什么将price变量声明为可以为null的十进制数值。

4.2.1 null条件运算符的连接运算

null条件运算符可以连在一起以遍历整个对象的层次结构，这使它成为简化代码以及进行安全导航的有效工具。在代码清单4-6中，在嵌套引用的Product类中添加一个属性，创建更加复杂的对象层次结构。

代码清单4-6 在Models文件夹下的Product.cs文件中添加一个属性

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }
```

```
        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

每一个Product对象都有一个可以引用其他Product对象的Related属性。在GetProducts方法中，设置代表皮艇的Product对象的Related属性。代码清单4-7展示了如何利用null条件运算符检验对象属性但不引起错误。

代码清单4-7　检测在Controllers文件夹下的HomeController.cs文件中嵌套的null值

```
using Microsoft.AspNetCore.Mvc;
```

```
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()
) {
                string name = p?.Name;
                decimal? price = p?.Price;
                string relatedName = p?.Related?.Name;
                results.Add(string.Format("Name: {0}, P
rice: {1}, Related: {2}",
                    name, price, relatedName));
            }

            return View(results);
        }
    }
}
```

null条件运算符可以被应用于属性链中的任意部分，就像下面这样：

```
...
string relatedName = p?.Related?.Name;
...
```

赋值逻辑是这样的：当p为null或者p.Related为null时，relatedName为null；否则，relatedName为p.Related.Name属性的值。运行程序，你将在浏览器窗口中看到如下输出：

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related:
Name: , Price: , Related:
```

4.2.2 联合使用null条件运算符和null合并运算符

通过将null条件运算符（单个问号）和null合并运算符（两个问号）联合起来使用，可以有效地将null值反馈给应用程序，如代码清单4-8所示。

代码清单4-8　在Controllers文件夹下的HomeController.cs文件中结合使用null条件运算符和null合并运算符

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

```
        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()
) {
                string name = p?.Name ?? "<No Name>";
                decimal? price = p?.Price ?? 0;
                string relatedName = p?.Related?.Name ?
? "<None>";
                results.Add(string.Format("Name: {0}, P
rice: {1}, Related: {2}",
                    name, price, relatedName));
            }
                return View(results);
        }
    }
}
```

null条件运算符可以保证在使用对象属性的过程中不会抛出NullReferenceException异常，而null合并运算符可以保证在浏览器的显示结果中不包含null值。运行程序，你将看到浏览器窗口中显示如下结果：

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related: <None>
Name: <No Name>, Price: 0, Related: <None>
```

4.3　使用自动实现属性

C#支持自动实现属性，我们之前在定义Person类的属性时使用过这项功能，如下所示：

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

这项功能使我们可以在定义属性的时候不必实

现get和set主体。使用自动实现属性让我们可以像如下示例这样定义属性：

```
...
public string Name { get; set; }
...
```

以上代码等效于以下代码：

```
...
public string Name {
    get { return name; }
    set { name = value; }
}
...
```

这种语法特征称为语法糖（syntactic sugar），这使得C#的使用变得更加友好。在本例中，你可以通过为赋值属性消除冗余代码而无须大规模地修改代码的运行逻辑。任何可以使代码变得更加易于编写和维护的优化都是有益的，特别是在大型项目中。

4.3.1　初始化自动实现的属性

自动实现的属性在C# 3.0以上的版本中都支持，可以在不需要构造函数的情况下设置初始值，如代码清单4-9所示。

代码清单4-9　在Models文件夹下的Product.cs文件中初始化自动实现的属性

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersp
orts";
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;
```

```
            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

为自动实现的属性赋值并不会阻止后续的setter方法对属性值进行改变，但是在需要使用构造函数为大量默认属性赋值的情况下，可以让代码更加简洁。代码清单4-9已将Water Craft赋给Category属性，并且初始值是可以更改的。

4.3.2　创建只读的自动实现属性

我们还可以创建只读属性，在初始化自动实现的属性时忽略 set 关键字即可实现这一点，如代码清单 4-10所示。

代码清单4-10　在Models文件夹下的Product.cs文件中创建只读的自动实现属性

```
namespace LanguageFeatures.Models {
```

```
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersp
orts";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; } = true;

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

InStock属性被初始化为true并且不能更改，但是可以在类的构造函数中进行赋值，如代码清单4-11所示。

代码清单4-11　在Models文件夹下的Product.cs文件中为只读的自动实现属性赋值

```
namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
             InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersp
orts";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }
        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };

            Product lifejacket = new Product(false) {
                Name = "Lifejacket",
                Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
```

```
}
```

构造函数允许将只读属性的值指定为参数，如果没有提供值，默认为true。在使用构造函数设置之后，属性值就无法更改了。

4.4　使用字符串插值

string.Format方法是构造包含数据值的字符串的传统C#工具。如下是Home控制器中的一个示例：

```
...
results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                          name, price, relatedName));
...
```

C# 6.0增加了对另外一种方式（称为字符串插值）的支持，这样就避免了字符串模板中的{0}引用必须与参数变量匹配。字符串插值直接使用变量

名就可以了，如代码清单4-12所示。

代码清单4-12　在Controller文件夹下的HomeController.cs文件中使用字符串插值

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()
) {
                string name = p?.Name ?? "<No Name>";
                decimal? price = p?.Price ?? 0;
                string relatedName = p?.Related?.Name ?
? "<None>";
                results.Add($"Name: {name}, Price: {pri
ce}, Related: {relatedName}");
            }

            return View(results);
        }
    }
}
```

插值字符串以$作为前缀并且包含槽（hole），这是对包含在{and}字符内的值的引用。计算字符串时，可利用指定的变量或常量来填充这些槽。

Visual Studio提供了用于创建插值字符串的智能感知支持，当键入{符号时，提供了可用方法的列表；这就最大限度减少了人为的输入错误，并且结果的字符串格式更易理解。

提　示

字符串插值支持string.Format方法可用的所有指定格式。格式说明符是槽的一部分，因此$"Price:{price:C2}将把price值格式化为带两位小数的货币值。

4.5　使用对象和集合初始化器

当在Product类的静态方法GetProducts中创建一个对象时，使用对象初始化器可以实现创建对象和初始化属性值一步完成，如下所示：

```
...
Product kayak = new Product {
    Name = "Kayak",
    Category = "Water Craft",
    Price = 275M
};
...
```

这是可以使C#语言易于使用的另一个语法糖特征。没有这个功能，我们就不得不调用Product构造函数，然后使用新创建的对象来设置每个属性的值，像下面这样：

```
...
Product kayak = new Product();
kayak.Name = "Kayak";
```

```
kayak.Category = "Water Craft";
kayak.Price = 275M;
...
```

类似的还有集合初始化器，利用它可以实现创建集合和初始化集合内容一步完成。例如，创建一个字符数组，如果没有初始化器，将需要指定数组的大小以及每一个数组元素的值，如代码清单4-13所示。

代码清单4-13　在Controllers文件夹下的HomeController.cs文件中初始化一个对象

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            string[] names = new string[3];
            names[0] = "Bob";
            names[1] = "Joe";
            names[2] = "Alice";
            return View("Index", names);
        }
    }
```

```
}
```

使用初始化器可以使设定数组元素的值成为所构造集合的一部分，这种结构隐式地为编译器提供了数组的大小，如代码清单4-14所示。

代码清单4-14　在Controllers文件夹下的HomeController.cs文件中使用初始化器

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View("Index", new string[] { "Bob",
"Joe", "Alice" });
        }
    }
}
```

数组元素被包含在“{”和“}”之间，这使得集合的定义更加简明并且可以实现集合的嵌套定义。代码清单4-14中的代码与代码清单4-13中的代码有相

同的效果，如果运行示例应用程序，你将在浏览器窗口中看到如下内容：

```
Bob
Joe
Alice
```

使用索引初始化器

C#梳理了集合初始化器的用法，可使用索引来创建字典这样的集合。代码清单4-15展示了如何通过传统的C#方式重写Index操作方法来初始化一个字典。

代码清单4-15　在Controllers文件夹下的HomeController.cs文件中初始化一个字典

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
```

```
            Dictionary<string, Product> products = new
Dictionary<string, Product> {
                { "Kayak", new Product { Name = "Kayak"
, Price = 275M } },
                { "Lifejacket", new Product{ Name = "Li
fejacket", Price = 48.95M } }
            };
            return View("Index", products.Keys);
        }
    }
}
```

初始化此类集合的语法过于依赖“{”和“}”了，尤其是在利用对象初始化器来定义集合值的时候。C#编译器支持使用一种更加自然的方法来初始化索引集合，这种方法与在初始化集合后检索或更改值是一致的，如代码清单4-16所示。

代码清单4-16　在Controllers文件夹下的HomeController.cs文件中使用C#集合初始化器语法

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

```
        public ViewResult Index() {
            Dictionary<string, Product> products = new
Dictionary<string, Product> {
                ["Kayak"] = new Product { Name = "Kayak
", Price = 275M },
                ["Lifejacket"] = new Product { Name = "
Lifejacket", Price = 48.95M }
            };

            return View("Index", products.Keys);
        }
    }
}
```

以上操作的效果等同于创建一个字典，键为Kayak和Lifejacket，值为Product对象，但元素是通过用于其他集合操作的索引表示法创建的。如果运行示例应用程序，在浏览器窗口中将可以看到如下结果：

```
Kayak
Lifejacket
```

4.6　模式匹配

最近对C#最有用的补充之一是对模式匹配的支

持，模式匹配可用于测试对象是特定类型的还是具有特定特征。另一种形式是语法糖，它可以极大地简化复杂的条件语句块。is关键字用于执行类型测试，如代码清单4-17所示。

代码清单4-17　对Controllers文件夹下的HomeController.cs文件执行类型测试

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            object[] data = new object[] { 275M, 29.95M,

            "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                if (data[i] is decimal d) {
                    total += d;
                }
            }

            return View("Index", new string[] { $"Total
: {total:C2}" });
```

```
        }
    }
}
```

is关键字执行类型检查，如果值是指定的类型，就将值赋给新变量，如下所示：

```
...
if (data[i] is decimal d) {
...
```

如果data[*i*]中存储的是十进制值，那么上述表达式的计算结果将为true。data[*i*]的值将被分配给变量*d*，这样就可以在随后的语句中使用该值，而无须执行任何类型转换。is关键字将只匹配指定的类型，这意味着只处理data数组中的两个值（数组中的其他元素是string和int值）。

如果运行应用程序，浏览器窗口会显示以下输出：

```
Total: $304.95
```

switch语句中的模式匹配

模式匹配也可以在switch语句中使用，switch语句支持when关键字，用于在case语句匹配值时进行限制，如代码清单4-18所示。

代码清单4-18　Controllers文件夹下的HomeController.cs文件中的模式匹配

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            object[] data = new object[] { 275M, 29.95M
,
                "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                switch (data[i]) {
                    case decimal decimalValue:
                        total += decimalValue;
                        break;
                    case int intValue when intValue > 5
0:
```

```
                        total += intValue;
                        break;
                }
            }

            return View("Index", new string[] { $"Total
: {total:C2}" });
        }
    }
}
```

要匹配特定类型的任何值，请在case语句中使用类型和变量名，如下所示：

```
...
case decimal decimalValue:
...
```

以上case语句能匹配任何十进制值，并将其赋给名为decimalValue的新变量。

为了更有选择性，可以包含when关键字，如下所示：

```
...
case int intValue when intValue > 50:
...
```

以上case语句能匹配int值，并将它们赋给名为intValue的变量，但仅当值大于50时才这么做。如果运行应用程序，浏览器窗口会显示以下输出：

```
Total: $404.95
```

4.7 使用扩展方法

扩展方法是在类中添加不属于用户且不能直接修改的方法的简便方式。代码清单4-19展示了ShoppingCart类的定义，将它添加到Models文件夹的ShoppingCart.cs文件中，用于表示Product对象的集合。

代码清单4-19 Models文件夹下的ShoppingCart.cs文件的内容

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart {
        public IEnumerable<Product> Products { get; set
; }
```

```
    }
}
```

这个简单的类可作为Product对象列表的封装器（对于本例我们只需要一个基础类）。假设我们需要定义ShoppingCart类中Product对象的总值，但是不能改变类本身，由于它来自第三方，因此我们可能无法获取源代码。这时候我们就可以使用扩展方法来添加我们需要的功能。代码清单4-20展示了添加到Models文件夹中的MyExtensionMethods类。

代码清单4-20　Models文件夹下的MyExtensionMethods.cs文件的内容

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this Shopping
Cart cartParam) {
            decimal total = 0;
            foreach (Product prod in cartParam.Products
) {
                total += prod?.Price ?? 0;
            }
```

```
            return total;
        }
    }
}
```

第一个参数前面的this关键字表明TotalPrices是一个扩展方法，用于告诉.NET在这种情况下可以对ShoppingCart类使用扩展方法。我们可以使用cartParam参数引用扩展方法作用的ShoppingCart实例。这里的方法枚举了ShoppingCart中的Product并且返回Product.Price属性的总值。代码清单4-21显示了如何在Home控制器的操作方法中应用扩展方法。

注　意

扩展方法不能打破类为自身的方法、字段和属

性定义的访问规则。可以利用扩展方法扩展类的功能，但是只能使用允许访问的类成员。

代码清单4-21　在Controllers文件夹下的HomeController.cs文件中使用扩展方法

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            ShoppingCart cart
            = new ShoppingCart { Products = Product.Get
Products() };
            decimal cartTotal = cart.TotalPrices();
            return View("Index", new string[] { $"Total
: {cartTotal:C2}" });
        }
    }
}
```

关键的声明语句如下：

```
...
decimal cartTotal = cart.TotalPrices();
...
```

在ShoppingCart对象中调用TotalPrices方法时就像这个方法是ShoppingCart类的成员一样，尽管它是一个由不同类定义的扩展方法。如果扩展类在当前类的作用域内，.NET将可以查找到它们，因为它们在同一个命名空间中，或者在using语句指定的命名空间中。如果运行示例程序，浏览器窗口会显示以下输出：

```
Total: $323.95
```

4.7.1　将扩展方法应用于接口

我们也可以创建应用于接口的扩展方法，这允许我们在实现接口的所有类中调用它们。代码清单4-22展示了更新的ShoppingCart类，它实现了IEnumerable<Product>接口。

代码清单4-22　在Models文件夹下的ShoppingCart.cs文件中实现一个接口

```
using System.Collections;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart : IEnumerable<Product> {
        public IEnumerable<Product> Products { get; set
; }

        public IEnumerator<Product> GetEnumerator() {
            return Products.GetEnumerator();
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }
}
```

现在可以更新扩展方法，以便处理IEnumerable<Product>，如代码清单4-23所示。

代码清单4-23　在Models文件夹下的MyExtensionMethods.cs文件中更新一个扩展方法

```
using System.Collections.Generic;
```

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumera
ble<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }
    }
}
```

第一个参数的类型被更改为IEnumerable<Product>，也就是说，方法体中的每一次for循环都直接作用在Product对象上。使用接口意味着可以计算由任何IEnumerable<Product>枚举的Product对象的总值，其中既包括ShoppingCart实例，也包含Product对象数组，如代码清单4-24所示。

代码清单4-24　在Controllers文件夹下的HomeController.cs文件中对数组使用扩展方法

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
        public ViewResult Index() {

            ShoppingCart cart
                = new ShoppingCart { Products = Product
.GetProducts() };

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
 = 48.95M}
            };

            decimal cartTotal = cart.TotalPrices();
            decimal arrayTotal = productArray.TotalPric
es();

            return View("Index", new string[] {
                $"Cart Total: {cartTotal:C2}",
                $"Array Total: {arrayTotal:C2}" });
        }
    }
}
```

如果运行示例程序，你将看到如下结果，结果表明不管Product对象是如何收集的，我们都能从扩

展方法中获得相同的结果：

```
Cart Total: $323.95
Array Total: $323.95
```

4.7.2　创建过滤扩展方法

扩展方法可以用于筛选对象的集合。作用在IEnumerable<T>上并且返回IEnumerable<T>的扩展方法可以使用yield关键字将选择条件应用于源数据中的项，以生成结果集。代码清单4-25演示了这样一种扩展方法，已将其添加到MyExtensionMethods类中。

代码清单4-25　在Controller文件夹下的MyExtensionMethods.cs文件中添加过滤扩展方法

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumera
```

```
ble<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }
        public static IEnumerable<Product> FilterByPric
e(
                this IEnumerable<Product> productEnum,
decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice)
 {
                    yield return prod;
                }
            }
        }
    }
}
```

FilterByPrice扩展方法采用一个附加参数来帮助筛选那些Price属性与参数相匹配或高于参数的Product对象，代码清单4-26展示了具体用法。

代码清单4-26　在Controllers文件夹下的HomeController.cs文件中使用过滤扩展方法

```
using Microsoft.AspNetCore.Mvc;
```

```
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
 = 48.95M},
                new Product {Name = "Soccer ball", Pric
e = 19.50M},
                new Product {Name = "Corner flag", Pric
e = 34.95M}
            };

            decimal arrayTotal = productArray.FilterByP
rice(20).TotalPrices();

            return View("Index", new string[] { $"Array
 Total: {arrayTotal:C2}" });
        }
    }
}
```

当我们对Product对象数组调用FilterByPrice方法时，TotalPrices方法只返回那些价格高于20美元的值，用于计算总值。如果运行示例程序，浏览器

窗口会显示如下输出：

```
Total: $358.90
```

4.8　使用Lambda表达式

Lambda 表达式会导致很多混淆，Lambda表达式本身极具简化的特性就会产生一种迷惑。为了理解要解决的问题，我们沿用之前定义的FilterByPrice扩展方法。此扩展方法用来根据价格筛选Product对象，也就是说，如果还要通过姓名进行筛选，那么需要再定义一个过滤扩展方法，如代码清单4-27所示。

代码清单4-27　在Models文件夹下的MyExtensionMethods.cs文件中添加一个过滤扩展方法

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {
```

```
        public static decimal TotalPrices(this IEnumera
ble<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByPric
e(
                this IEnumerable<Product> productEnum,
decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice)
 {
                    yield return prod;
                }
            }
        }

        public static IEnumerable<Product> FilterByName(

                this IEnumerable<Product> productEnum,
char firstLetter) {

            foreach (Product prod in productEnum) {
                if (prod?.Name?[0] == firstLetter) {
                    yield return prod;
                }
            }
        }
    }
}
```

代码清单4-28展示了如何在控制器中使用两种过滤扩展方法来创建两个不同的总量。

代码清单4-28　在Controllers文件夹下的HomeController.cs文件中使用两种过滤扩展方法

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
 = 48.95M},
                new Product {Name = "Soccer ball", Pric
e = 19.50M},
                new Product {Name = "Corner flag", Pric
e = 34.95M}
            };

            decimal priceFilterTotal = productArray.Fil
terByPrice(20).TotalPrices();
            decimal nameFilterTotal = productArray.Filt
erByName('S').TotalPrices();
```

```
            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}
```

第一个过滤扩展方法选择价格为20美元以上的所有商品，第二个过滤扩展方法选择名称以字母S开头的商品。如果运行示例程序，你将在浏览器窗口中看到以下输出：

```
Price Total: $358.90
Name Total: $19.50
```

4.8.1　定义函数

可以不断重复以上过程，并为感兴趣的每一个属性和属性组合创建不同的过滤扩展方法。更优雅的方式是将处理枚举的代码从选择条件中分离出来。用C#处理这个比较简单，C#通过允许函数作为对象传递实现了这一操作。代码清单4-29展示了

一个扩展方法，用于过滤Product对象的枚举，但将选择结果中元素的决策操作委托给了一个单独的函数。

代码清单4-29　在Models文件夹下的MyExtensionMethods.cs文件中创建常规的Filter方法

```
using System.Collections.Generic;
using System;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumera
ble<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }

        public static IEnumerable<Product> Filter(
                this IEnumerable<Product> productEnum,
                Func<Product, bool> selector) {

            foreach (Product prod in productEnum) {
                if (selector(prod)) {
                    yield return prod;
```

```
                }
            }
        }
    }
}
```

Filter方法的第二个参数是一个接收Product对象并返回bool值的函数。Filter方法为每个Product对象调用该函数，如果返回true，就将Product对象添加到结果中。要使用Filter方法，可以指定另一个方法或创建独立的函数，如代码清单4-30所示。

代码清单4-30　在Controllers文件夹下的HomeController.cs文件中使用函数来过滤Product对象

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        bool FilterByPrice(Product p) {
            return (p?.Price ?? 0) >= 20;
        }

        public ViewResult Index() {
```

```
            Product[] productArray = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
 = 48.95M},
                new Product {Name = "Soccer ball", Pric
e = 19.50M},
                new Product {Name = "Corner flag", Pric
e = 34.95M}
            };

            Func<Product, bool> nameFilter = delegate (
Product prod) {
                return prod?.Name?[0] == 'S';
            };

            decimal priceFilterTotal = productArray
                .Filter(FilterByPrice)
                .TotalPrices();
            decimal nameFilterTotal = productArray
                .Filter(nameFilter)
                .TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}
```

但这两种方式都不理想。FilterByPrice这样的方法会扰乱类的定义。创建Func<Product,bool>对象

虽然可以避免这个问题，但其笨拙的语法难以阅读和维护。Lambda表达式可以解决这个问题，从而以一种更加优雅且更具表现力的方式来定义函数，如代码清单4-31所示。

代码清单4-31　在Controllers文件夹下的HomeController.cs文件中使用Lambda表达式

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
 = 48.95M},
                new Product {Name = "Soccer ball", Pric
e = 19.50M},
                new Product {Name = "Corner flag", Pric
e = 34.95M}
            };
```

```
            decimal priceFilterTotal = productArray
                .Filter(p => (p?.Price ?? 0) >= 20)
                .TotalPrices();
            decimal nameFilterTotal = productArray
                .Filter(p => p?.Name?[0] == 'S')
                .TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}
```

在代码清单4-31中，Lambda表达式以粗体显示。参数的类型不需要指定，可自动推断。符号“=>”连接了参数和Lambda表达式的结果。在示例程序中，名为p的Product参数被连接到bool型结果。如果Price属性的值大于或等于20美元（第一个表达式），或者Name属性以S开头（第二个表达式），那么参数p为true。虽然代码的工作方式与单独的方法和函数代理相同，但它更简洁，对于大多数人来说易读性更强。

Lambda表达式的其他形式

不需要在Lambda表达式中表达委托的逻辑。我们可以很容易地调用方法，就像下面这样：

```
prod => EvaluateProduct(prod)
```

如果在委托函数中需要具有多个参数的Lambda表达式，就必须将这些参数包含在括号中，如下所示：

```
(prod, count) => prod.Price > 20 && count > 0
```

最后，如果需要在具有多个语句的Lambda表达式中使用逻辑，那么可以换用大括号（{}），并使用return语句返回结果，如下所示：

```
(prod, count) => {
    // <em>...multiple code statements...</em>
    return result;
```

```
}
```

不需要在代码中使用Lambda表达式，但它们确实是表达复杂函数的一种简单清晰的方式。

4.8.2　使用Lambda表达式实现方法和属性

在C# 6.0中，对Lambda进行了扩展以便它们可以实现方法和属性。在MVC开发中，特别是在编写控制器时，通常会使用包含单个语句的方法来选择要显示的数据和要呈现的视图。在代码清单4-32中，重写了Index操作方法。

代码清单4-32　在Controllers文件夹下的HomeController.cs文件中创建公共action模式

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
```

```
namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View(Product.GetProducts().Select(p
=> p?.Name));
        }
    }
}
```

操作方法Index将从静态的Product.GetProducts方法中获取Product对象的集合，并使用LINQ生成Name属性的值，然后用作默认视图的视图模型。如果运行示例程序，你将在浏览器窗口中看到以下输出：

```
Kayak
Lifejacket
```

浏览器窗口中也会有一个空的列表项，因为GetProducts方法在结果中包含了null引用，但这对本章的这一部分并不重要。

当一个方法的主体由单个语句组成时，可以将其重写为Lambda表达式，如代码清单4-33所示。

代码清单4-33　在Controllers文件夹下的HomeController.cs文件中将操作方法重写为Lambda表达式

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() =>
        View(Product.GetProducts().Select(p => p?.Name)
);
    }
}
```

方法的Lambda表达式省略了return关键字，并使用=>将方法签名（包括参数）与其实现相关联。代码清单4-33所示的Index操作方法的工作方式与代码清单4-32中的相同，但表达式更简洁。这种

基本方法也可以用来定义属性。代码清单4-34显示了如何在Product类中使用Lambda表达式来添加属性。

代码清单4-34　在Models文件夹下的Product.cs文件中将属性表示为Lambda表达式

```
namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
            InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersp
orts";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }
        public bool NameBeginsWithS => Name?[0] == 'S';

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
```

```
            Product lifejacket = new Product(false) {
                Name = "Lifejacket",
                Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, n
ull };
        }
    }
}
```

4.9 使用类型推断和匿名类型

C#的var关键字允许你在不显式指定变量类型的情况下定义局部变量，如代码清单4-35所示，这称为类型推断（type inference）或隐式类型。

代码清单4-35 在Controller文件夹下的HomeController.cs文件中使用类型推断

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
```

```
namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
        public ViewResult Index() {
            var names = new [] { "Kayak", "Lifejacket",
 "Soccer ball" };
            return View(names);
        }
    }
}
```

names变量并不是没有类型，相反，要求编译器从代码中推断出类型。编译器检查数组声明，并推断出names是一个字符串数组。运行示例程序，将生成以下输出：

```
Kayak
Lifejacket
Soccer ball
```

使用匿名类型

通过组合对象的初始化器和类型推断，可以创建简单的视图模型对象，它们对于在控制器和视图之间传输数据很有用，而不必定义类或结构，如代

码清单4-36所示。

代码清单4-36　在Controller文件夹下的HomeController.cs文件中创建匿名类型

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.9
5M },
                new { Name = "Soccer ball", Price = 19.
50M },
                new { Name = "Corner flag", Price = 34.
95M }
            };

            return View(products.Select(p => p.Name));
        }
    }
}
```

products数组中的每个对象都是匿名类型的对

象，但这并不是说在这种情况下JavaScript变量是动态的，而只是意味着编译器将自动创建类型定义。强制类型转换仍在执行。例如，可以获取并设置在初始化器中定义的属性。如果运行示例程序，你将在浏览器窗口中看到以下输出：

```
Kayak
Lifejacket
Soccer ball
Corner flag
```

C#编译器根据初始化器中参数的名称和类型来生成类。两个具有相同属性名和类型的匿名类型对象将被分配给同一个自动生成器类。这意味着Product数组中的所有对象都将具有相同的类型，因为它们拥有相同的属性。

提　示

必须使用var关键字定义匿名类型对象的数组，由于在代码被编译之前不会指定类型，因此不知道要使用哪种类型。匿名类型对象数组中的元素必须拥有相同的属性，否则，编译器无法确定数组类型应该是什么。

为了证明这一点，更改代码清单4-37的输出，以便显示类型名称而不是Name属性的值。

代码清单4-37　在Controllers文件夹下的HomeController.cs文件中显示Name属性的类型

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
```

```
    public class HomeController : Controller {

        public ViewResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.9
5M },
                new { Name = "Soccer ball", Price = 19.
50M },
                new { Name = "Corner flag", Price = 34.
95M }
            };

            return View(products.Select(p => p.GetType(
).Name));
        }
    }
}
```

products数组中的所有对象都被分配相同的类型，类型名称并不友好并且不能直接使用，你可能会看到与以下输出不同的名字：

```
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
```

4.10　使用异步方法

C#中最近添加的一项重要内容是对异步方法（asynchronous method）的处理方式的改进。异步方法会在后台进行，并在完成后通知你，这样在执行后台工作时代码就能够处理其他事务。异步方法是从代码中消除瓶颈，并允许应用程序利用多个处理器和处理器内核并行执行工作的重要工具。

在MVC中，异步方法可以提高应用程序的总体性能，这是因为服务器在调度和执行请求的方式上有了更大的灵活性。C#关键字async和await用于异步工作的执行。

为此，需要在示例项目中添加一个新的.NET程序集，以便可以进行异步HTTP请求。在Solution Explorer窗格中右击LanguageFeatures文件夹，在弹出的菜单中选择Edit LanguageFeatures.csproj，添加一些内容，如代码清单4-38所示。

代码清单4-38　在LanguageFeatures文件夹下的LanguageFeatures.csproj文件中添加程序集引用

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version
="4.3.2" />
  </ItemGroup>

</Project>
```

在保存LanguageFeatures.csproj文件后，Visual Studio将下载System.Net.Http程序集并将其添加到项目中。第6章将更详细地描述这一过程。

4.10.1　直接使用任务

C#和.NET对异步方法有很好的支持，但代码往往十分冗长，并且不习惯并行编程的开发人员通常会因为不同的语法而感到困惑。例如，代码清单4-39显示了MyAsyncMethods类中定义的GetPageLength异步方法，可将其添加到MyAsyncMethods.cs类文件。

代码清单4-39　Models文件夹下的MyAsyncMethods.cs文件的内容

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public static Task<long?> GetPageLength() {

            HttpClient client = new HttpClient();

            var httpTask = client.GetAsync("http://apre
ss.com");

            return httpTask.ContinueWith((Task<HttpResp
onseMessage> antecedent) => {
                return antecedent.Result.Content.Header
```

```
s.ContentLength;
                });
            }
        }
}
```

上述代码使用一个System.Net.Http.HttpClient对象请求Apress主页的内容并返回内容的长度。.NET代表将被异步完成的任务。Task对象根据后台工作产生的结果强制进行类型化。所以，当调用HttpClient.GetAsync方法时，得到的是一个Task<HttpResponseMessage>对象。这说明请求将在后台执行，并且返回的结果是一个HttpResponseMessage对象。

提　示

当使用“后台运行”这样的字眼时，表示跳过很多细节，只聚焦MVC相关的关键点。.NET对异步方法和并行编程的支持是非常好的，如果要创建能够利用多核和多处理器硬件的真正高性能的应用程序，建议你进一步了解它。随着更多功能被介绍，你将看到MVC如何使创建异步Web应用程序变得容易。

在本例中，使用ContinueWith方法处理从HttpClient返回的对象。在GetAsync方法中，使用一个Lambda表达式在HttpResponseMessage中返回一个属性的值，其中包含从Apress Web服务器获取的内容的长度。

```
...
return httpTask.ContinueWith((Task<HttpResponseMessage>
 antecedent) => {
    return antecedent.Result.Content.Headers.ContentLen
gth;
```

```
});
...
```

请注意，以上代码使用了两次return关键字。第一次使用return关键字是为了指定返回一个Task <HttpResponseMessage>对象，当任务完成时，将返回ContentLength头部的长度。ContentLength头部会返回一个可为null的long值，这意味着GetPageLength方法的返回结果是Task <long?>，如下所示：

```
...
public static Task<long?> GetPageLength() {
...
```

不要介意这种方式是否有道理，也不要担心，因为很多人也会遇到同样的问题，为此微软在C#中添加了两个关键字来简化异步方法的使用。

4.10.2 使用async和await关键字

C#引入了两个关键字，专门用于简化异步方法（如HttpClient.GetAsync）的使用。这两个关键字是async和await，代码清单4-40展示了怎样应用它们来简化异步方法的使用。

代码清单4-40　在Models文件夹下的MyAsyncMethods.cs文件中使用async和await关键字

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength()
 {

            HttpClient client = new HttpClient();

            var httpMessage = await client.GetAsync("ht
tp://apress.com");

            return httpMessage.Content.Headers.ContentL
ength;
        }
    }
}
```

在调用异步方法时使用了await关键字，从而告诉C#编译器要等待GetAsync方法返回Task结果，然后继续执行同一方法中的其他语句。

应用await关键字意味着可以处理从GetAsync方法返回的Task结果，尽管GetAsync只是常规方法并且只返回给HttpResponseMessage对象一个值。最重要的是，可以通过正常的方式使用其他方法的return关键字返回结果，在本例中也就是ContentLength属性的值。这是一种顺理成章的技术，意味着我们不必担心ContinueWith方法和return关键字的多次使用。

使用await关键字时，还必须将async关键字添加到方法签名中，就像在示例中所做的那样。方法的返回结果类型不变。示例中的GetPageLength方法仍然返回Task <long？>，这是因为await和async关键字是使用一些巧妙的编译技巧实现的：允许使用

一种更自然的语法，但并不会改变应用它们的方法要执行的操作。在调用GetPageLength方法时仍然要处理Task <long？>结果，因为仍有后台操作会生成可为null的long值，当然，程序员也可以选择使用await和async关键字。

以上模式将贯穿于整个MVC控制器，这使得编写异步操作方法变得简单，如代码清单4-41所示。

代码清单4-41　在Controllers文件夹下的HomeController.cs文件中定义异步操作方法

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public async Task<ViewResult> Index() {
            long? length = await MyAsyncMethods.GetPage
```

```
Length();
            return View(new string[] { $"Length: {lengt
h}" });
        }
    }
}
```

这里已将Index操作方法的结果修改为Task<ViewResult>，从而告诉MVC该操作方法将返回一个Task对象并在完成时生成一个ViewResult对象，提供渲染视图所需的详细信息和数据。这里为方法定义添加了async关键字，这样就可以在调用MyAsyncMethods.GetPathLength方法时使用await关键字了。MVC和.NET负责处理后续事项，从而得到易于编写、易于阅读、易于维护的异步代码。如果运行该应用程序，你将看到类似于下面的输出内容（可能长度是不同的，因为Apress网站的内容经常更改）：

```
Length: 54576
```

4.11　获取名称

在Web应用开发中有许多任务需要引用参数、变量、方法或类的名称，比如处理用户输入时引发的异常或者创建验证错误。传统的方法是使用硬编码的方式获得名称，如代码清单4-42所示。

代码清单4-42　在Controller文件夹下的HomeController.cs文件中对名称进行硬编码

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.9
5M },
                new { Name = "Soccer ball", Price = 19.
50M },
                new { Name = "Corner flag", Price = 34.
```

```
95M }
            };

            return View(products.Select(p => $"Name: {p
.Name}, Price: {p.Price}"));
        }
    }
}
```

调用LINQ Select方法将生成一个字符串序列，其中的每一个字符串都包含对Name和Price属性的硬编码引用。运行应用程序会在浏览器窗口中生成以下输出：

```
Name: Kayak, Price: 275
Name: Lifejacket, Price: 48.95
Name: Soccer ball, Price: 19.50
Name: Corner flag, Price: 34.95
```

这种方法存在的问题是容易出现错误，要么因为名称键入有误，要么因为代码被重构，而字符串中的名称却没有正确更新。结果容易产生误导，特别容易出现问题。C#引入了nameof表达式，由编译器负责生成名称字符串，如代码清单4-43所示。

代码清单4-43　在Controllers文件夹下的HomeController.cs文件中使用nameof表达式

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
        public ViewResult Index() {

            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.9
5M },
                new { Name = "Soccer ball", Price = 19.
50M },
                new { Name = "Corner flag", Price = 34.
95M }
            };

            return View(products.Select(p =>
                $"{nameof(p.Name)}: {p.Name}, {nameof(p
.Price)}: {p.Price}"));
        }
    }
}
```

编译器通过形如p.Name的方式处理引用，以便

仅将最后一部分包含在字符串中，从而生成与前面示例中相同的输出。Visual Studio提供了nameof表达式的智能感知支持，可提示你选择引用，并且在重构代码时正确地更新表达式。由于编译器负责处理nameof表达式，因此使用无效引用会导致编译错误，从而可以收到错误引用或过期引用的通知。

4.12 小结

本章概述了一名合格的MVC程序员需要了解的关键C#特征。C#是一种非常灵活的语言，提供各种不同的方法来处理各种问题，但这些问题又是Web应用开发过程中经常会遇到的，这在本书的很多示例中都有体现。下一章将介绍Razor视图引擎，并解释如何将之应用于MVC Web应用程序中以生成动态内容。

第5章　使用Razor

在ASP.NET Core MVC应用程序中，视图引擎（view engine）负责处理发送给客户端的内容。MVC框架中默认的视图引擎称为Razor，用来为HTML文件添加注释说明并将这些动态内容插入发送给浏览器的输出中。

本章将介绍一个用于快速解读Razor语法的工具，这样当你看到它的时候，就可以马上将其识别出来。本章不会非常详细地介绍Razor的内容，把本章看作Razor语法的速成指南即可。随着本书其他章节介绍MVC的其他功能，我们再慢慢深入介绍。关于在具体语境中如何理解Razor，参见表5-1。

表**5-1**　在具体语境中理解**Razor**

问题	答案
它是什么？	Razor是负责将数据合并到HTML文档中的视图引擎
它有什么作用？	动态生成内容的能力对编写Web应用程序是必不可少的，Razor提供了可以使C#语句轻松地与ASP.NET Core MVC的其他部分协同工作的能力
怎样使用？	将Razor表达式添加到视图文件的静态HTML中，Razor表达式可协助生成客户端请求的响应
有哪些容易出现的问题或限制？	Razor表达式可以包含几乎所有的C#语句，并且很难决定逻辑应该属于视图还是控制器，从而削弱MVC关注点分离的核心理念
有其他的做法吗？	可以编写自己的视图引擎。也有一些第三方的视图引擎可以使用，但它们往往在特定情况下可行，不提供长期支持

表5-2列出了本章要完成的操作。

表**5-2**　本章要完成的操作

操作	解决方法	代码清单
访问视图模型	使用@model表达式定义模型类型，使用@Model表达式访问模型对象	代码清单5-5、代码清单5-14、代码清单5-17
使用类型名称而不限定它们的命名空间	创建视图导入文件	代码清单5-6和代码清单5-7

定义多视图共享的内容	使用布局	代码清单5-8～代码清单5-10
定义默认布局	使用视图启动文件	代码清单5-11～代码清单5-13
将数据从控制器传递到视图模型之外的其他视图	使用视图包	代码清单5-15和代码清单5-16
有选择地生成内容	使用Razor条件表达式	代码清单5-18和代码清单5-19
为数组和集合中的每个元素生成内容	使用Razor的foreach表达式	代码清单5-20和代码清单5-21

5.1　准备示例项目

为了演示Razor是怎样工作的，首先创建一个ASP.NET Core Web Application（.NET Core）项目并取名为Razor，就像之前的章节描述的那样。接下来，在Startup.cs文件中启用了MVC的默认配置，如代码清单5-1所示。

代码清单5-1　在Razor文件夹下的Startup.cs文件中启用MVC的默认配置

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Razor {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }
            //app.Run(async (context) => {
            //    await context.Response.WriteAsync("He
llo World!");
            //});
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

5.1.1　定义模型

接下来，创建模型文件夹Models，并在其中添加名为Product.cs的类文件，用于定义代码清单5-2中的简单模型类。

代码清单5-2　Models文件夹下的Product.cs文件的内容

```
namespace Razor.Models {

    public class Product {

        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

5.1.2　创建控制器

Startup.cs文件中的配置如下：默认情况下，MVC将请求发送到名为Home的控制器。创建Controllers文件夹，并在其中添加名为

HomeController.cs的类文件，用于定义代码清单5-3中的简单控制器。

代码清单5-3　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            Product myProduct = new Product {
                ProductID = 1,
                Name = "Kayak",
                Description = "A boat for one person",
                Category = "Watersports",
                Price = 275M
            };

            return View(myProduct);
        }
    }
}
```

以上控制器定义了一个名为Index的操作方法来创建和填充Product对象的属性。将Product对象传

递给View方法，以便渲染视图时将其用作模型。在调用View方法时，并不指定视图文件的名称，这样操作方法便会使用默认视图。

5.1.3　创建视图

为了给 Index 操作方法创建默认视图，创建Views/Home文件夹，在其中添加MVC视图页面文件Index.cshtml，其中的内容如代码清单5-4所示。

代码清单5-4　Views/Home文件夹下的Index.cshtml文件的内容

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
```

```
    Content will go here
</body>
</html>
```

后面将介绍Razor视图的其他部分，并演示其中一些内容的不同作用。在学习Razor的时候，请记住一点，视图的存在是为了向用户表达模型的一个或多个方面。也就是说，可利用从一个或多个对象中检索到的数据来生成HTML 页面文件。如果你仍记得我们始终试图建立可以发送到客户端的HTML页面，那么就会觉得Razor所做的一切都是有意义的。如果运行示例应用程序，你将看到图5-1所示的输出。

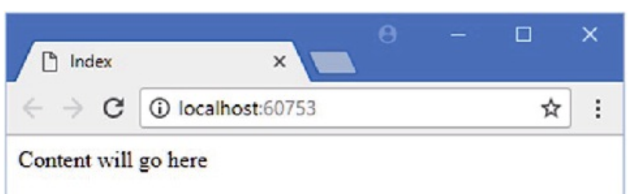


图5-1　示例应用程序的输出

5.2　使用模型对象

让我们从Index.cshtml视图文件的第一行开始。

```
...
@model Razor.Models.Product
...
```

Razor表达式以@字符开头。在本例中，@model表达式声明了将从操作方法传递到视图的模型对象的类型。这样就可以通过@model来访问视图模型对象的方法、字段和属性，代码清单5-5显示了对Index视图所做的简单补充。

代码清单5-5　在Views/Home文件夹下的Index.cshtml文件中引用视图模型对象的属性

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
```

```
    <title>Index</title>
</head>
<body>
    @Model.Name
</body>
</html>
```

注　意

这里使用@model声明视图模型对象的类型，而使用@Model访问Name属性。

如果运行应用程序，你将看到图5-2所示的输出。

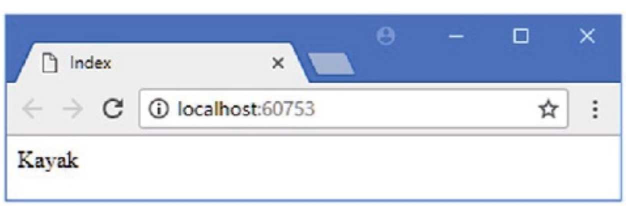


图5-2　在视图中读取属性的输出

使用@model表达式指定类型的视图称为强制类型视图（strongly typed view）。当键入@model和句点时，Visual Studio便会弹出@model表达式成员可以使用的名称建议，如图5-3所示。

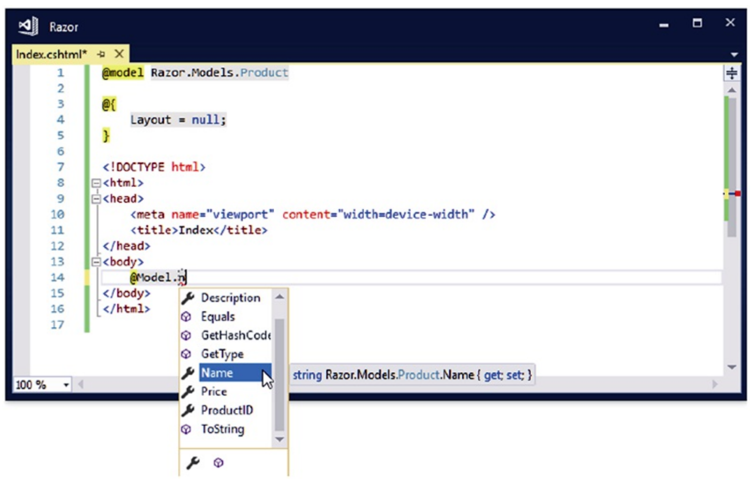


图5-3　Visual Studio基于@model表达式为成员名称提供建议

Visual Studio对成员名称的可视化建议有助于避免在Razor中出现错误。根据个人意愿，可以忽略这些建议，Visual Studio将高亮显示成员名称有问题，以便进行更正，就像使用常规的C#类文件一

样。图5-4列举了一个示例，开发人员在其中试图引用@Model.NotARealProperty。Visual Studio已意识到开发人员在模型类型中指定的Product类没有这样的属性，因而在编辑器中高亮显示错误。



图5-4　Visual Studio为@Model表达式报告错误

使用视图导入

当在Index.cshtml文件的开头定义模型对象时，必须导入包含模型类的命名空间，如下所示：

```
...
@model Razor.Models.Product
...
```

默认情况下，在强制类型的Razor视图中引用的所有类型都必须使用命名空间进行限定。当模型

对象有唯一的引用类型时，这不是什么大问题；但是当需要编写更复杂的Razor表达式时，就会使视图的可读性变差。

可以通过在项目中添加视图导入文件来指定要搜索类型的命名空间集合。视图导入文件放在Views文件夹中，并命名为_ViewImports.cshtml。

注　意

Views文件夹中名称以下画线（_）开头的文件不会返回给用户，从而将想要呈现的视图文件和支持它们的文件区分开。视图导入文件和布局模板（稍后介绍）是以下画线为前缀的。

要创建视图导入文件，请在解决方案资源管理器（Solution Explorer）中右击Views文件夹，从弹出菜单中选择Add→New Item，然后从ASP.NET类别中选择MVC View Imports Page模板，如图5-5所示。

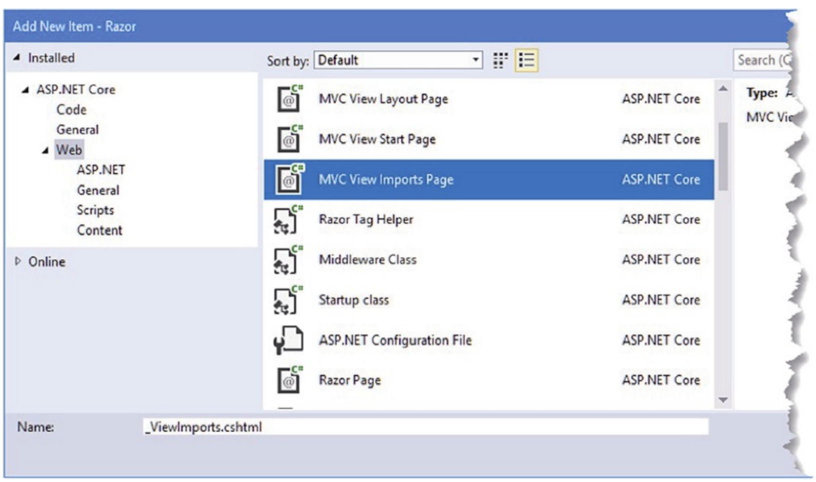


图5-5　创建视图导入文件

Visual Studio会自动将文件的名称设置为_ViewImports. cshtml，单击Add按钮以创建该文件。代码清单5-6展示了添加完表达式之后的视

图。

代码清单5-6　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using Razor.Models
```

Razor视图中用来搜索类的命名空间由@ using表达式指定，后面跟着相应的命名空间。在代码清单5-6中，已为Razor.Models命名空间添加了一项，其中包含了示例应用程序的模型类。

现在，Razor.Models命名空间就包含在视图导入文件中了，可以从Index.cshtml文件中移除命名空间，如代码清单5-7所示。

代码清单5-7　在Views/Home文件夹下的Index.cshtml文件中不使用命名空间引用模型类

```
@model Product
@{
```

```
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
</head>
<body>
    @Model.Name
</body>
</html>
```

提　示

还可以将@using表达式添加到单个视图文件中，这将允许在单个视图中使用没有命名空间的类型。

5.3　使用布局

Index.cshtml视图文件中还有另外一个重要的Razor表达式：

```
...
@{
    Layout = null;
}
...
```

这是一个Razor代码块，允许在视图中包含C#语句。Razor代码块以@“{”开始，以“}”结束，里面包含的语句将在呈现视图时执行。

以上Razor代码块将Layout属性的值设置为null。Razor视图被编译为MVC应用程序中的C#类，并且用基类定义了Layout属性。第21章将介绍具体的工作方式。将Layout属性设置为null的效果就是告诉MVC视图是自包含的，并将渲染客户端所需的所有内容。

自包含的视图对于简单的示例应用程序表现比较好，但是一个真正的项目可以有几十个视图，有些视图将共享内容。在视图中复制共享内容又很难管理，尤其是当需要进行更改并且必须跟踪所有需要更改的视图时。

当一个模板包含公共的内容并且可以应用于一个或多个视图时，比较好的方法就是使用Razor布局。对布局进行更改时，更改将自动影响使用布局的所有视图。

5.3.1　创建布局

布局通常由多个控制器使用的视图共享，并存储在Views/Shared文件夹中，这是在查找文件时Razor会自动查看的位置之一。为了创建布局，先创建Views/Shared文件夹，右击后从弹出菜单中选择Add→New Item。从ASP.NET类别中选择MVC

View Layout Page模板，并将文件名设置为_BasicLayout.cshtml，如图5-6所示。单击Add按钮创建这个文件（与视图导入文件一样，布局文件的名称也以下画线开头）。

代码清单5-8显示了由Visual Studio创建的_BasicLayout.cshtml文件的初始内容。

代码清单5-8　Views/Shared文件夹下的_BasicLayout.cshtml文件中的初始内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

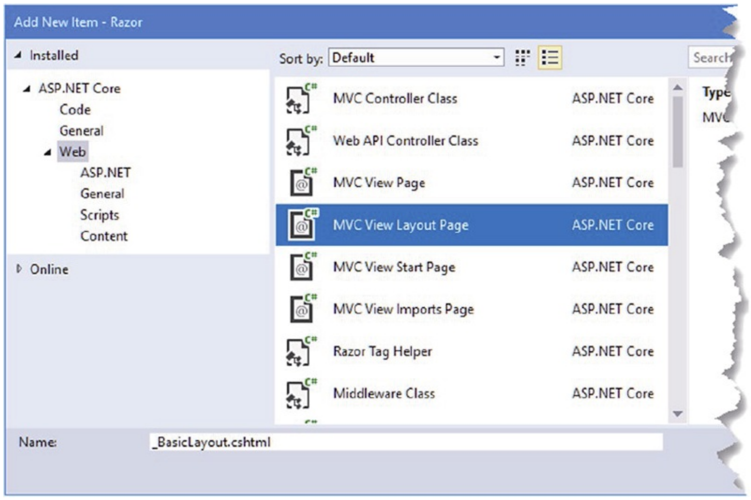


图5-6　创建布局

布局是一种特殊的视图形式，代码中已高亮显示@表达式。@RenderBody方法的调用会将操作方法定义的视图内容插入布局标记中。

```
...
<div>
    @RenderBody()
</div>
...
```

布局中的另一个Razor表达式会查找名为ViewBag.Title的属性，以便设置title元素的内容。

```
...
```

```
<title>@ViewBag.Title</title>
...
```

ViewBag允许在应用程序中传递数据值，本例中是指在视图和布局之间传递。在将布局应用于视图时，你将看到这是如何工作的。

布局中的HTML元素将应用于任何使用它们的视图，并且提供了一个用于定义公共内容的模板。在代码清单5-9中，在布局中添加了一些简单的标记，这样模板效果就很明显了。

代码清单5-9　在Views/Shared文件夹下的_BasicLayout.cshtml文件中添加内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <style>
        #mainDiv {
            padding: 20px;
            border: solid medium black;
```

```
            font-size: 20pt
        }
    </style>
</head>
<body>
    <h1>Product Information</h1>
    <div id="mainDiv">
        @RenderBody()
    </div>
</body>
</html>
```

这里添加了标头元素以及一些CSS来为包含@RenderBody表达式的div元素的内容设置样式，这样就可以清楚地知道哪些内容来自布局，哪些内容来自视图。

5.3.2　使用布局

要将布局应用于视图，我们需要设置Layout属性的值，并移除现在由布局提供的HTML，如代码清单5-10所示的html、head和body元素。

代码清单5-10　在Views/Home文件夹下的Index.cshtml文件中

```
@model Product

@{
    Layout = "_BasicLayout";
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

Layout属性将指定用于视图的布局文件，但不使用.cshtml文件扩展名。Razor将在/Views/Home和Views/Shared文件夹中查找指定的布局文件。

上面还设置了ViewBag.Title属性。在渲染视图的时候，布局会使用该属性设置title元素的内容。

即使是非常简单的应用，视图的这种转换也是非常激动人心的。布局包含了任何HTML响应所需的所有结构，这使得视图只需要关注向用户呈现数据的动态内容就可以了。当MVC处理Index.cshtml文件时，将应用布局来创建统一的HTML响应，如

图5-7所示。

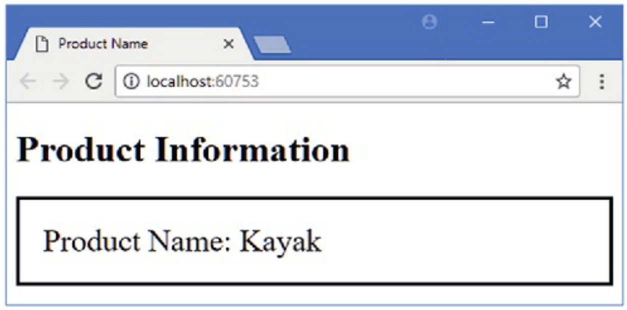


图5-7　为视图使用布局后的效果

5.3.3　应用视图启动文件

另一个需要交代的地方，就是必须为将要使用的每个视图指定布局文件。因此，如果需要重命名布局文件，就必须找到引用布局文件的每个视图并进行更改，这非常容易出错。最重要的是，这与MVC应用程序的易维护宗旨背道而驰。

这个问题可以通过使用视图启动文件（view start file）来解决。在渲染视图的时候，MVC将查找一个名为_ViewStart.cshtml的文件。这个文件的

内容将被视为包含在视图文件本身中，我们可以使用这个功能自动设置Layout属性的值。

要创建视图启动文件，请右击Views文件夹，从弹出菜单中选择Add→New Item，然后从ASP.NET类别中选择MVC View Start Page模板，如图5-8所示。

Visual Studio会自动将文件的名称设置为_ViewStart.cshtml，单击Add按钮，Visual Studio为这个文件创建的初始内容如代码清单5-11所示。

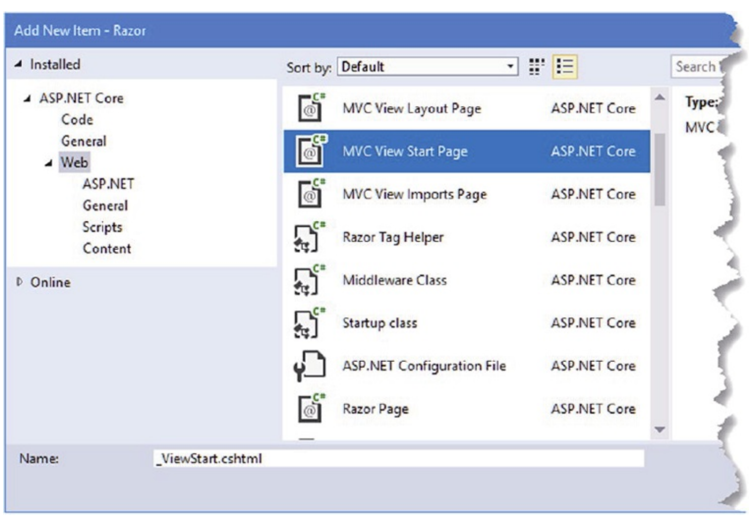


图5-8　创建视图启动文件

代码清单5-11　Views文件夹下的_ViewStart.cshtml文件的初始内容

```
@{
    Layout = "_Layout";
}
```

为了将布局应用于应用程序中的所有视图，更改分配给Layout属性的值，如代码清单5-12所示。

代码清单5-12　在Views文件夹下的_ViewStart.cshtml文件中应用默认视图

```
@{
    Layout = "_BasicLayout";
}
```

因为视图启动文件包含Layout属性的值，所以可以从Index.cshtml文件中删除相应的表达式，如代码清单5-13所示。

代码清单5-13　更新Views/Home文件夹下的Index.cshtml文件以体现视图启动文件的使用

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

不必指定要使用的视图启动文件。MVC将自动定位视图启动文件并使用里面的内容。视图启动文件中定义的值优先，这样就可以方便地重写视图启动文件。

还可以使用多个视图启动文件为应用程序的不同部分设置默认值。Razor会查找离正在处理的视图最近的视图启动文件，这意味着可以通过在Views/Home或Views/Shared文件夹中添加视图启动文件来覆盖默认设置。



警　告

务必了解从视图启动文件中省略Layout属性与将Layout属性设置为null之间的区别。如果视图是自包含的，并且不希望使用布局，可将Layout属性设置为nulll。如果省略Layout属性，MVC将认为需要布局，并且应该使用已在视图启动文件中找到的值。

5.4　使用Razor表达式

至此，我们已经了解了视图和布局的基本知识，接下来介绍Razor支持的各种不同类型的表达式，以及如何使用它们来创建视图内容。在优秀的MVC应用程序中，操作方法和视图的角色之间存在着明确的差异，如表5-3所示。

表**5-3**　操作方法和视图的角色差异

内容	可以做什么	不能做什么
操作方法	将视图模型对象传递到视图	将格式化数据传递到视图
视图	使用视图模型对象向用户显示内容	修改视图模型对象

为了充分发挥MVC的优势，我们需要尊重并保证应用程序不同部分之间的分离。正如你将看到的，Razor可以为我们做很多工作，包括使用C#语句，但是不能使用Razor执行业务逻辑或以任何方式操作域模型对象。

作为一个简单的示例，代码清单5-14显示了一种在Index.cshtml文件中添加表达式的方式。

代码清单5-14　在Views/Home文件夹下的Index.cshtml文件中添加一个表达式

```
@model Product
```

```
@{
    ViewBag.Title = "Product Name";
}

<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
```

你可以在操作方法中设置Price属性的值，并将其传递给视图。虽然这样可以奏效，但是采用这种方法会破坏MVC模式自身的优点，并降低对未来变化的响应能力。虽然ASP.NET Core MVC不强制正确使用MVC模式，但你应该了解不这么做可能带来的影响。

处理数据与格式化数据

区分处理数据和格式化数据是非常重要的。视图会格式化数据而不是将对象的属性格式化为字符串，这就是之前将Product对象传递给视图的原因。

处理数据（包括选择要显示的数据对象）是控制器的职责，控制器将调用模型来获取和修改所需的数据。有时很难弄清楚处理和格式化之间的界限是什么，但作为一条经验法则，谨慎起见，建议把除了最简单的表达式以外的任何东西从视图中拿出来移到控制器中。

5.4.1　插入数据

使用Razor表达式可以轻松将数据插入标记中。最常见的方法是使用@Model表达式，Index视图已包含具体示例，如下所示：

```
...
<p>Product Name: @Model.Name</p>
...
```

还可以使用ViewBag功能插入数据，也就是之

前在布局中用来设置title元素内容的那项功能。ViewBag可用于将数据从控制器传递到视图以补充模型，如代码清单5-15所示。

代码清单5-15　在Controller文件夹下的HomeController.cs文件中使用ViewBag功能

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {
        public ViewResult Index() {
            Product myProduct = new Product {
                ProductID = 1,
                Name = "Kayak",
                Description = "A boat for one person",
                Category = "Watersports",
                Price = 275M
            };

            ViewBag.StockLevel = 2;

            return View(myProduct);
        }
    }
}
```

ViewBag属性会返回一个可用于定义任意属性

的dynamic对象。在代码清单5-15中，定义了一个StockLevel属性，并将属性值指定为2。由于ViewBag是动态的，因此不必事先声明属性名称，但这意味着Visual Studio无法为ViewBag属性提供自动完成的建议。

何时使用ViewBag以及何时扩展模型是经验和个人偏好问题。作者喜欢只使用ViewBag来提供说明如何呈现数据的视图提示，而非用于显示给用户的数据。如果确实要将视图包用于显示给用户的数据，可以使用@ViewBag表达式来访问它们，如代码清单5-16所示。

代码清单5-16　在Views/Home文件夹下的Index.cshtml文件中显示视图包

```
@model Product

@{
    ViewBag.Title = "Product Name";
}
```

```
<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
<p>Stock Level: @ViewBag.StockLevel</p>
```

图5-9显示了运行结果。

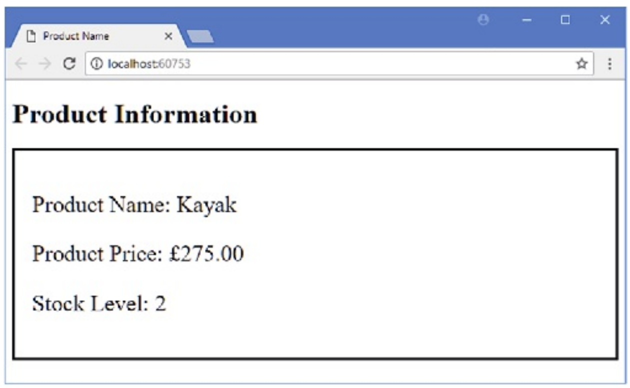


图5-9　使用Razor表达式插入数据的结果

5.4.2　设置属性值

到目前为止，已为所有的示例设置了元素的内容，但也可以使用Razor表达式来设置元素的属性值。代码清单5-17显示了如何使用@Model和@ViewBag表达式来设置Index视图中元素的属性值。

代码清单5-17　在Views/Home文件夹下的Index.cshtml文件中使用Razor表达式来设置属性值

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel=
"@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level: @ViewBag.StockLevel</p>
</div>
```

这里使用Razor表达式设置了div元素的data属性值。

提　示

名称由data-作为前缀的属性，多年来一直是一种创建自定义属性的非正式方法，而今天作为HTML5的一部分已成为正式标准的一部分。它们经常被用到，所以JavaScript代码可以定位特定的元素，并且CSS样式可以得到更精准的应用。

如果运行示例程序并查看发送到浏览器的HTML源代码，你将看到Razor已经设置了属性的值：

```
<div data-productid="1" data-stocklevel="2">
    <p>Product Name: Kayak</p>
    <p>Product Price: £275.00</p>
    <p>Stock Level: 2</p>
</div>
```

5.4.3　使用条件语句

Razor能够处理条件语句，也就是说，我们可

以根据视图数据中的值调整视图的输出。这种技术是Razor的核心，使得使用者可以创建复杂但流畅的布局，同时又易读、易于维护。在代码清单5-18中，修改Index视图以包含条件语句。

代码清单5-18　在Views/Home文件夹下的Index.cshtml文件中使用Razor条件语句

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel=
"@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level:
        @switch (ViewBag.StockLevel) {
            case 0:
                @:Out of Stock
                break;
            case 1:
            case 2:
            case 3:
                <b>Low Stock (@ViewBag.StockLevel)</b>
                break;
            default:
```

```
                @: @ViewBag.StockLevel in Stock
                break;
        }
    </p>
</div>
```

为了使用条件语句，请在C#条件关键字的前面放置@字符，如示例中的@switch。就像使用常规的C#代码块一样，代码块的终止符号为右大括号“}”。

在Razor代码块内，可以通过定义HTML和Razor表达式将HTML元素和数据包含在视图输出中，如下所示：

```
...
<b>Low Stock (@ViewBag.StockLevel)</b>
...
```

元素或表达式不需要放在引号中或者以特殊的方式表示它们——Razor引擎会将它们解释为要处理的输出。但是，如果要在视图中不包含HTML元

素时向其中插入文本，就需要对Razor进行特殊标注，并将所在行的前缀写成如下形式：

```
...
@: Out of Stock
...
```

@:字符是为了防止Razor被解释为C#语句，这是遇到文本时的默认行为。你可以在图5-10中看到条件语句的结果。

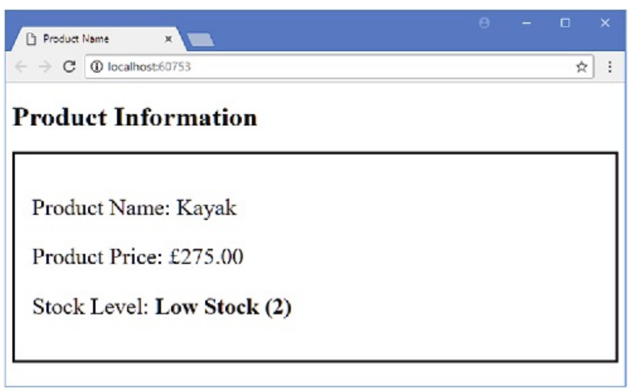


图5-10　在Razor视图中使用开关表达式

条件语句在Razor视图中很重要，因为它们允许根据视图从操作方法接收的数据来对内容进行更改。在这里我们添加一些额外的演示，代码清单 5-

19 演示了如何在Index.cshtml文件中添加if语句。

代码清单5-19　在Views/Home文件夹下的Index.cshtml文件的Razor视图中使用if语句

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel=
"@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level:
        @if (ViewBag.StockLevel == 0) {
            @:Out of Stock
        } else if (ViewBag.StockLevel > 0 && ViewBag.St
ockLevel <= 3) {
            <b>Low Stock (@ViewBag.StockLevel)</b>
        } else {
            @: @ViewBag.StockLevel in Stock
        }
    </p>
</div>
```

这个条件语句的结果与switch语句相同，这里演示的是如何让Razor视图与C#条件语句相匹配。

第21章将会解释这里的条件语句是怎样工作的。

5.4.4　枚举数组和集合

在编写MVC应用程序时，通常需要枚举数组的内容或某些其他类型对象的集合，并针对每个元素内容执行相应的操作。为了演示这是如何完成的，在代码清单5-20中，修改Home控制器中的Index操作方法，以便将一个Product对象数组传递给视图。

代码清单5-20　在Controllers文件夹下的HomeController.cs文件中使用数组

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            Product[] array = {
                new Product {Name = "Kayak", Price = 27
5M},
                new Product {Name = "Lifejacket", Price
```

```
 = 48.95M},
                new Product {Name = "Soccer ball", Pric
e = 19.50M},
                new Product {Name = "Corner flag", Pric
e = 34.95M}
            };
            return View(array);
        }
    }
}
```

Index操作方法创建了一个包含简单数据的Product[]对象，并将它们传递给View方法，以便使用默认视图呈现数据。在代码清单5-21中，更改Index视图的模型类型，并使用foreach循环枚举数组中的每个对象。

提　示

代码清单5-21中的Model不需要以@字符作为

前缀，因为它是C#表达式的一部分。我们很难弄清楚何时需要使用@字符，但是当不需要使用时，Visual Studio智能感知功能会通过下画线提示有错误。

代码清单5-21　在Views/Home文件夹下的Index.cshtml文件中枚举数组

```
@model Product[]

@{
    ViewBag.Title = "Product Name";
}

<table>
    <thead>
        <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody>
        @foreach (Product p in Model) {
            <tr>
                <td>@p.Name</td>
                <td>@($"{p.Price:C2}")</td>
            </tr>
        }
    </tbody>
```

```
</table>
```

@foreach语句用于枚举Model数组的内容，并为每个数组元素生成一行数据。以上代码在foreach循环中创建了名为p的局部变量，然后使用Razor表达式@p.Name与@p.Price引用属性名称和价格，结果如图5-11所示。



图5-11　使用Rzaor枚举数组的结果

5.5　小结

本章概述了Razor视图引擎以及如何用它生成HTML，介绍了如何通过视图模型对象和ViewBag来引用从控制器传来的数据，以及如何使用Razor

表达式根据数据定制对用户的响应。接下来的章节将介绍更多关于Razor用法的示例，第21章将详细介绍MVC视图的机制。下一章将介绍Visual Studio为使用ASP.NET Core MVC项目而提供的一些功能。

第6章　使用Visual Studio

本章主要介绍Visual Studio为开发ASP.NET Core MVC项目提供的关键特性。表6-1列出了本章要完成的操作。

表**6-1**　本章要完成的操作

操作	方法	代码清单
添加包到项目中	使用NuGet工具管理.NET包，使用Bower管理客户端包	代码清单6-6～代码清单6-8
查看视图或类更改的效果	使用迭代开发模式	代码清单6-9～代码清单6-11
在浏览器中显示详细消息	使用开发人员调试页	代码清单6-12
获取有关程序执行的详细信息和控制信息	使用调试器	代码清单6-13
使用Visual Studio重新加载一个或多个浏览器	使用浏览器链接	代码清单6-14和代码清单6-15

减少HTTP请求的数量以及JavaScript和CSS文件所需的带宽大小	使用Bundler & Minifier扩展	代码清单6-16～代码清单6-23

6.1　准备示例项目

在本章中，将使用Empty模板创建一个新的名为WorkingWithVisualStudio的ASP.NET Core Web Application（.NET Core）项目。在Startup.cs文件中用默认配置启用MVC，如代码清单6-1所示。

代码清单6-1　在WorkingWithVisualStudio文件夹下的Startup.cs文件中启用MVC

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
namespace WorkingWithVisualStudio {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
```

```
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

6.1.1 创建模型

创建Models文件夹，并添加名为Product.cs的类文件，用来定义代码清单6-2中的类。

代码清单6-2 Models文件夹下的Product.cs文件的内容

```
namespace WorkingWithVisualStudio.Models {

    public class Product {
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

为了创建一个简单的Product存储对象，在Models文件夹中添加名为SimpleRepository.cs的类

文件，用来定义代码清单6-3中的类。

代码清单6-3　Models文件夹下的SimpleRepository.cs文件的内容

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
    public class SimpleRepository {
        private static SimpleRepository sharedRepositor
y = new SimpleRepository();
        private Dictionary<string, Product> products
            = new Dictionary<string, Product>();

        public static SimpleRepository SharedRepository
 => sharedRepository;

        public SimpleRepository() {
            var initialItems = new[] {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M },
                new Product { Name = "Corner flag", Pri
ce = 34.95M }
            };
            foreach (var p in initialItems) {
                AddProduct(p);
            }
        }
```

```
        public IEnumerable<Product> Products => products.Values;

        public void AddProduct(Product p) => products.Add(p.Name, p);
    }
}
```

这个类会将模型对象存储在内存中，这意味着当应用程序停止或重新启动时，对模型所做的任何更改都将丢失。非持久化存储对于本章中的示例来说已经足够，但对于许多实际项目来说并不可取。关于创建持久化存储模型对象的示例，请参见8章。

注　意

代码清单6-3定义了一个名为SharedRepository

的静态属性，使用它就可以在整个应用程序中对SimpleRepository对象进行访问。这虽然不是最好的做法，但可以通过它展示一下在MVC开发过程中经常会遇到的一个问题。第18章将介绍如何使用共享组件来更好地解决这个问题。

6.1.2　创建控制器和视图

在项目中创建Controllers文件夹，并添加名为HomeController.cs的类文件，用来定义代码清单6-4中的控制器。

代码清单6-4　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;

namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {
```

```
        public IActionResult Index()
            => View(SimpleRepository.SharedRepository.P
roducts);
    }
}
```

Index操作方法会获取所有模型对象并将它们传递给View方法以呈现默认视图。为了添加默认视图，创建Views/Home文件夹，在其中添加名为Index.cshtml的视图文件，里面的内容如代码清单6-5所示。

代码清单6-5　Views/Home文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<WorkingWithVisualStudio.Models.Produ
ct>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
</head>
<body>
    <table>
```

```
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@p.Price</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

Index视图中包含一个表格，使用Razor foreach循环为每个模型对象创建行，其中每一行都包含Name和Price属性。如果运行示例应用程序，你将看到图6-1所示的结果。

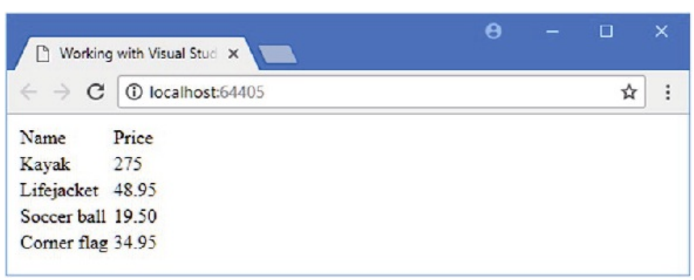


图6-1　运行示例应用程序的结果

6.2 管理软件包

ASP.NET Core MVC项目需要两种不同类型的软件包，后面将描述其中的每种类型以及Visual Studio为管理它们而提供的工具。

6.2.1 NuGet

为了管理包含在项目中的.NET包，Visual Studio提供了一个图形工具——NuGet。要打开该图形工具，请选择Tools→NuGet Package Manager→Manage NuGet Packages for Solution，弹出的界面如图6-2所示。

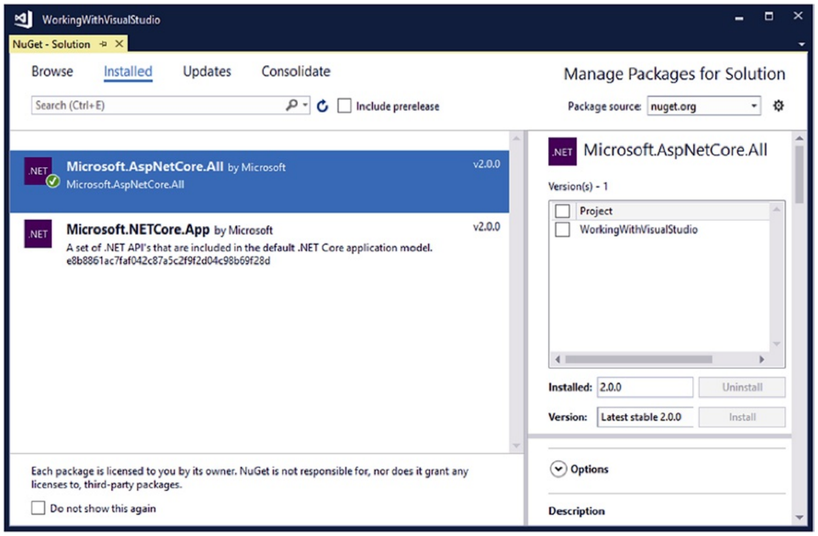


图6-2　弹出的界面

Installed选项卡提供了项目中已安装的包的摘要信息，Browse选项卡可用于定位和安装新的包，Updates选项卡可用于列出已发布的最新版本的软件包。

Microsoft.AspNetCore.All包

如果使用了早期版本的ASP.NET Core，你将明白向新项目添加大量NuGet包的必要性。ASP.NET

Core 2采用不同的方法，依赖一个名为Microsoft.AspNetCore.All的包。

Microsoft.AspNetCore.All包是元包（meta-package），里面包含了ASP.NET Core和MVC框架所需的所有单个NuGet包，这意味着不需要逐个添加包。在发布应用程序时，将删除元包中未被应用程序使用的单个包，以确保不部署其他没有用到的包。

NuGet包列表和位置

NuGet工具在<projectname>.csproj文件中提供了包的有关信息，其中<projectname>可由项目名称替换。对于示例应用程序，这意味着NuGet包的详细信息存储在名为WorkingWithVisualStudio.csproj

的文件中。Visual Studio不会在Solution Explorer窗格中显示.csproj文件。要编辑该文件，请在Solution Explorer窗格中右击项目，然后在弹出的菜单中选择Edit WorkingWithVisualStudio.csproj。Visual Studio将打开该文件进行编辑。.csproj文件是XML格式的，你将看到类似下面这样的元素，从而将ASP.NET Core元包添加到项目中：

```
...
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All"
Version="2.0.0" />
</ItemGroup>
...
```

为每个包指定名称和所需的版本号。尽管元包包括ASP.NET Core MVC所需的所有功能，但你仍必须将包添加到项目中，以便可以使用附加功能。可以使用图6-2所示的界面或使用命令行工具来添加软件包。也可以直接编辑.csproj文件，Visual

Studio会检测到更改并下载和安装相应的软件包。

当使用NuGet将一本包添加到项目中时，这个包将与它所依赖的所有包一起自动安装。可以通过在Solution Explorer窗格中选择Dependencies→NuGet来研究NuGet包及其依赖项。ASP.NET Core元包具有大量的依赖项，其中一些依赖项如图6-3所示。

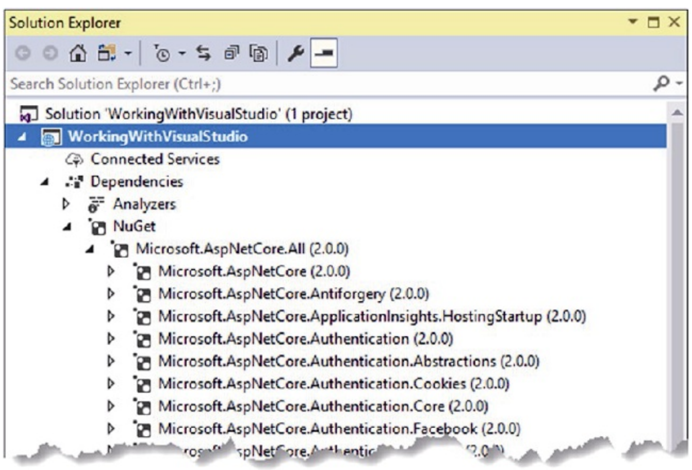


图6-3　Solution Explorer窗格中的一些依赖项

6.2.2　Bower

客户端包（client-side package）是包含发送到客户端的内容的包，如JavaScript文件、层叠样式表或图像。NuGet过去也用来管理这些项目，但是ASP.NET Core MVC依赖于Bower工具。Bower是一个开源工具，已经独立于Microsoft和.NET开发完成，并被广泛应用于非ASP.NET Web应用程序的开发中。

注　意

Bower最近已弃用。但是，Bower仍在积极维护中，并且对Bower的支持已集成到Visual Studio中。

在某些时候，可以期望Microsoft支持其他用于

管理客户端包的工具，但是在这种情况发生之前，应该继续使用Bower。

1．Bower包列表

Bower包可通过bower.json文件指定。要创建Bower配置文件，请在Solution Explorer窗格中右击Working WithVisualStudio项目，从弹出的菜单中选择Add→New Item，然后在左侧窗格中选择ASP.NET Core→Web→General，在右侧窗格中选择Bower Configuration File项目模板，如图6-4所示。

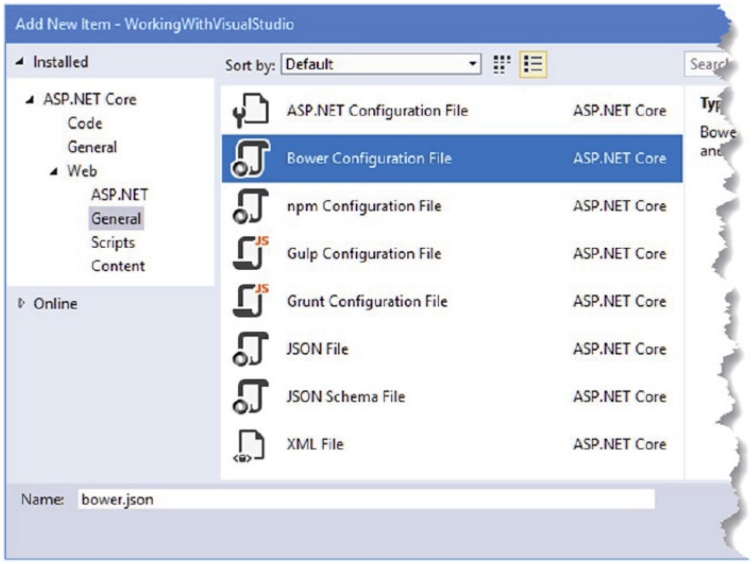


图6-4　创建Bower配置文件

Visual Studio已将名称设置为bower.json，单击Add按钮将该文件添加到项目中，默认内容如代码清单6-6所示。

代码清单6-6　bower.json文件的默认内容

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
  }
}
```

代码清单6-7显示了将客户端包添加到

bower.json文件中的方法，这是通过添加与project.json文件相同格式的项到dependencies部分来实现的。

提　示

Bower软件包的存储库参见Bower网站，可以在其中搜索要添加到项目中的软件包。

代码清单6-7　在bower.json文件中添加包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7"
  }
```

```
}
```

以上代码将Bootstrap CSS包添加到了示例项目中。当编辑bower.json文件时，Visual Studio将为你提供一个包名列表，并列出可用包的版本，如图6-5所示。

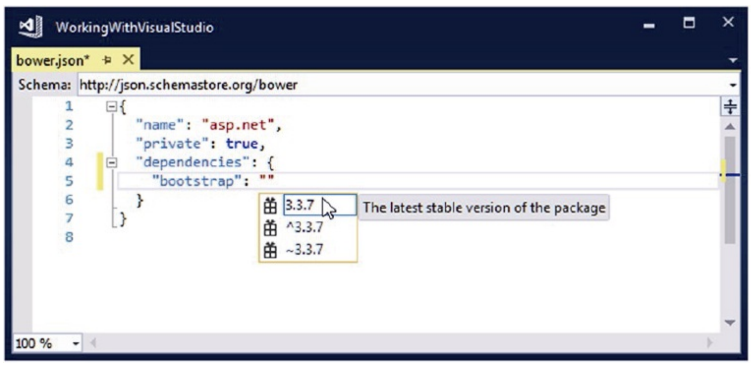


图6-5　列出客户端包的可用版本

在编写本书时，Bootstrap包的最新版本是3.3.7。但是请注意，Visual Studio提供了3个版本号，分别是3.3.7、^3.3.7和 ~3.3.7。在bower.json文件中，版本号可以不同的方式指定，一些常用模式如表6-2所示。指定包的最安全方法是使用显式的

版本号。这样就能确保始终使用相同的版本，除非特意更新bower.json文件来更换其他版本。

提　示

对于本书中的示例，直接创建和编辑.json文件。该文件易于编辑，有助于你得到期望的结果。Visual Studio为管理Bower软件包提供了一个图形工具，右击bower.json文件，从弹出的菜单中选择Manage Bower Package即可打开该工具。

表**6-2　bower.json**文件中版本号的一些常用模式

模式	描述

3.3.7	安装与给定的版本号完全匹配的软件包，例如3.3.7
*	使用星号将允许Bower下载和安装任何版本的软件包
>3.3.7 >=3.3.7	使用>或>=作为版本号的前缀将允许Bower安装大于以及大于或等于给定版本号的软件包的任何版本
<3.3.7 <=3.3.7	使用<或<=作为版本号的前缀将允许Bower安装小于以及小于或等于给定版本号的软件包的任何版本
~3.3.7	使用波浪符（~字符）作为版本号的前缀将允许Bower安装即使版本号的最后一位不匹配的版本。例如，指定~3.3.7将允许Bower安装3.3.8或3.3.9版本，但不能安装3.4.0版本
^3.3.7	使用插入符（^字符）作为版本号的前缀将允许Bower安装即使版本号的第二位或补丁号不匹配的版本。例如，指定^ 3.3.0将允许Bower安装3.3.1、3.4.0和3.5.0版本，但不可以安装4.0.0版本

Visual Studio会监控bower.json文件是否更改，并自动使用Bower工具下载和安装软件包。当把更改保存到代码清单6-7所示的文件时，Visual Studio将下载Bootstrap包并将其安装到wwwroot/lib文件夹中，如图6-6所示。

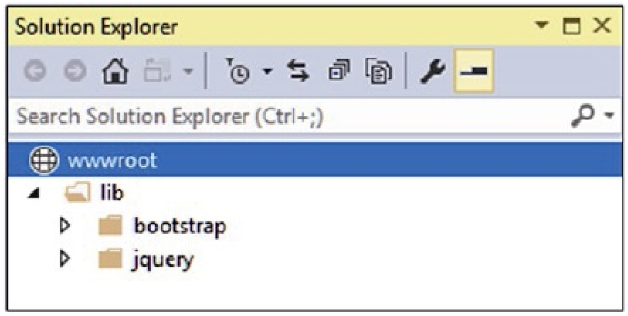


图6-6　为项目添加客户端包

就像NuGet一样，Bower可管理添加到项目中的包的依赖。对于一些高级功能，Bootstrap依赖jQuery JavaScript库，这就是为什么在图6-6中有两个包。展开Solution Explorer窗格中的依赖项，就可以查看包及其依赖项的列表，如图6-7所示。

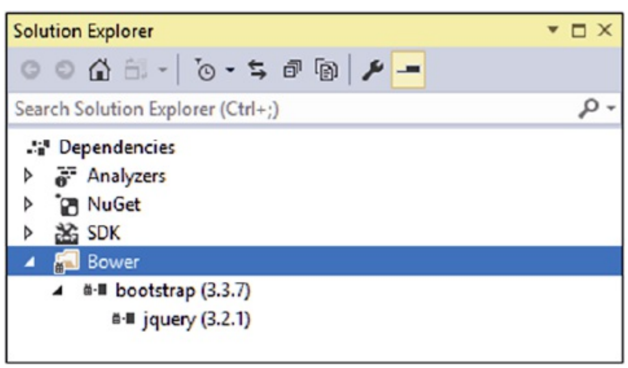


图6-7　检查客户端包及其依赖项

2．更新**Bootstrap**包

在本书的其余部分，使用了Bootstrap CSS框架的预发布版本。在撰写本书时，Bootstrap团队正在开发第4版，并且已经发布了几个早期版本。这些版本被标记为alpha，但是质量很高，并且它们足够稳定，可以在本书的示例中使用。当考虑是选择使用即将过时的Bootstrap 3还是使用Bootstrap 4的预发布版本来编写这本书时，最终作者决定使用新版本，尽管在最终发布之前，用于设置HTML元素样式的一些类名可能会更改。这意味着你必须使用相同版本的Bootstrap包才能从示例中获得预期的结果。

要更新Bootstrap包，请更改bower.json文件中的版本号，如代码清单6-8所示。

代码清单6-8　在WorkingWithVisualStudio文件夹下的bower.json文件中更改包的版本

```
{
```

```
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

在保存对bower.json文件所做的更改时，Visual Studio将下载新版的Bootstrap包。

6.3　迭代开发

Web应用开发通常是一个迭代过程，可以对视图或类进行细微的更改，并运行应用程序来测试它们的效果。让MVC和Visual Studio协同工作便可以实现这种迭代并方便快捷地看到更改效果。

6.3.1　修改Razor视图

在开发过程中，一旦收到来自浏览器的HTTP请求，对Razor视图所做的更改就会立即生效。为了演示这是如何工作的，请从Debug菜单中选择

Start Debugging，启动应用程序，打开浏览器并显示数据后，对Index.cshtml文件所做的更改如代码清单6-9所示。

代码清单6-9　更改Index.cshtml文件

```
@model IEnumerable<WorkingWithVisualStudio.Models.Produ
ct>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
</head>
<body>
    <h3>Products</h3>
    <table>
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@($"{p.Price:C2}")</td>
                </tr>
            }
        </tbody>
```

```
    </table>
</body>
</html>
```

将更改保存到Index视图中，然后单击浏览器中的Reload按钮以重新加载当前网页。对视图的更改（添加页眉以及将Price属性设置为美元格式）将生效并显示在浏览器中，如图6-8所示。

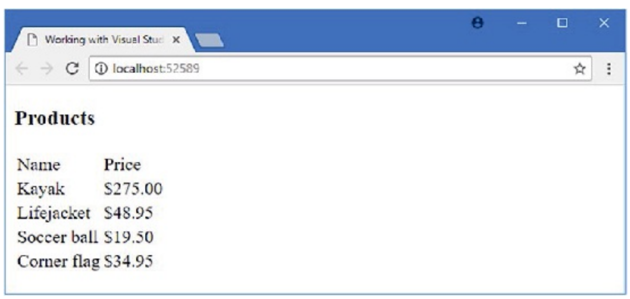


图6-8　对视图进行更改

提　示

第21章将介绍Razor视图的使用过程。

6.3.2　对C#类进行更改

对于C#类（包括控制器和模型），处理更改的方式取决于如何启动应用程序。后面会介绍两种可用方式，可以通过Debug菜单中的不同菜单项进行选择，如表6-3所示。

表6-3　**Debug**菜单项

菜单项	描述
Start Without Debugging	当接收到HTTP请求时，项目中的类将自动编译，这是一种更动态的开发体验。应用程序在没有调试器的情况下运行，所以不能控制代码的执行
Start Debugging	在这种开发模式下，必须显式地编译项目并重新启动应用程序以使更改生效。调试器在程序运行时添加到应用程序中，这样可以检查运行状态并分析任何问题

1．自动编译类

在正常开发过程中，快速迭代可以使你立即看到更改的效果，无论是添加新操作还是更改视图模型选择的数据都可以。对于此类开发，当浏览器接收到HTTP请求后，Visual Studio便能立即检测到更改并自动重新编译类。要查看此操作的工作方式，请从Visual Studio的Debug菜单中选择Start Without Debugging菜单项。在浏览器显示了应用数据后，对Home控制器的更改如代码清单6-10所示。

代码清单6-10　在HomeController.cs文件中筛选模型数据

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;
namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {

        public IActionResult Index()
            => View(SimpleRepository.SharedRepository.P
roducts
                        .Where(p => p.Price < 50));
    }
}
```

以上代码使用LINQ来筛选Product对象，只有Price属性小于50的才能被传递给视图。将更改保存到控制器类文件并重新加载浏览器窗口，而无须停止或重新启动Visual Studio中的应用程序。浏览器的HTTP请求将触发编译过程，应用程序将使用修改后的控制器类重新启动，从而产生图6-9所示的结果，其中省略了表格中的Kayak产品。

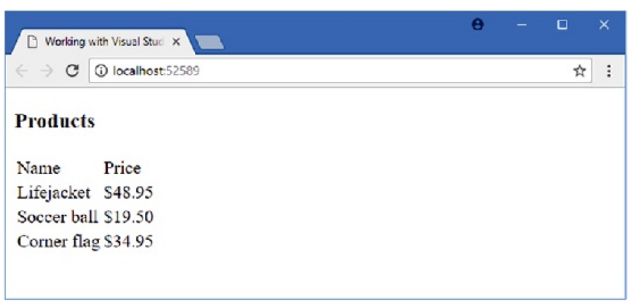


图6-9　自动编译类

当一切都按计划进行的时候，自动编译功能是非常好用的。但是当编译或运行过程中出现错误时，错误会直接显示在浏览器而不是Visual Studio中，此时我们便很难找出哪里出了状况。例如，代

码清单6-11显示了如何为存储库中的模型对象集合添加null引用。

代码清单6-11　在SimpleRepository.cs文件中添加null引用

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
    public class SimpleRepository {
        private static SimpleRepository sharedRepositor
y = new SimpleRepository();
        private Dictionary<string, Product> products
            = new Dictionary<string, Product>();

        public static SimpleRepository SharedRepository
 => sharedRepository;
        public SimpleRepository() {
            var initialItems = new[] {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M },
                new Product { Name = "Corner flag", Pri
ce = 34.95M }
            };
            foreach (var p in initialItems) {
                AddProduct(p);
            }
            products.Add("Error", null);
        }
```

```
        public IEnumerable<Product> Products => product
s.Values;

        public void AddProduct(Product p) => products.A
dd(p.Name, p);
    }
}
```

Visual Studio的智能感知功能将突出显示语法问题，但在应用程序运行之前，不会显示null引用这样的问题。重新加载浏览器中的页面时才会编译SimpleRepository类，并且应用程序会被重启。当MVC创建控制器类的实例以处理来自浏览器的HTTP请求时，HomeController构造函数才会实例化SimpleRepository类，这将反过来尝试处理在列表中添加的null引用。null值会引发一个问题，但此时还不清楚这个问题是什么，因为浏览器没有显示有用的消息。

2．启用开发人员异常页面

在开发过程中，当出现错误时，在浏览器窗口中显示更多有用的信息会大有帮助。可以通过启用开发人员异常页面来完成，我们需要对Startup类进行配置更改，如代码清单6-12所示。

代码清单6-12　在Startup.cs文件中启用开发人员异常页面

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

第14章将详细介绍Startup类的作用，但是现在只需要知道调用UseDeveloperExceptionPage扩展方法以设置错误描述页面就可以了。

重新加载浏览器窗口，自动编译过程将重新生成应用程序，并在浏览器中生成更有用的错误消息，如图6-10所示。

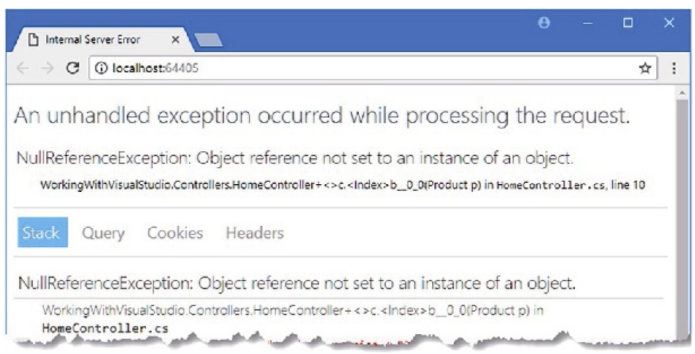


图6-10　一个开发人员异常页面中的错误消息

浏览器显示的错误消息足以解决简单的问题，比如由于迭代开发，最近新更改的内容可能是引起问题的原因。但是，对于复杂的问题或者那些不能立即显现的问题，需要使用Visual Studio调试器。

3．使用调试器

Visual Studio还支持使用调试器运行MVC应用程序，可以暂停执行以检查应用程序的状态和代码逻辑规定的请求应遵循的路径。这需要使用不同的开发模式，因为在重新启动应用程序之前对C#类所做的修改不会起作用（尽管对Razor视图所做的更改会自动生效）。

这种开发方式不像使用自动编译那样具有动态性，但是Visual Studio调试器非常好用，可以在浏览器窗口中显示信息，从而使开发人员对应用程序的工作方式有更深层次的了解。

要使用调试器运行应用程序，请从Visual Studio的Debug菜单中选择Start Debugging。Visual Studio将在启动应用程序之前编译项目中的C#类，也可以使用Build菜单手动编译代码。

示例应用程序中仍然包含null引用，也就是说，由SimpleRepository类抛出的未处理的NullReferenceException将中断应用程序并将执行控制传递给开发人员，如图6-11所示。

提　示

如果调试器没有截获异常，请从Visual Studio的菜单栏中选择Debug→Windows→Exception Settings并确保Common Language Runtime Exceptions列表中所有的异常类型都被检测到。

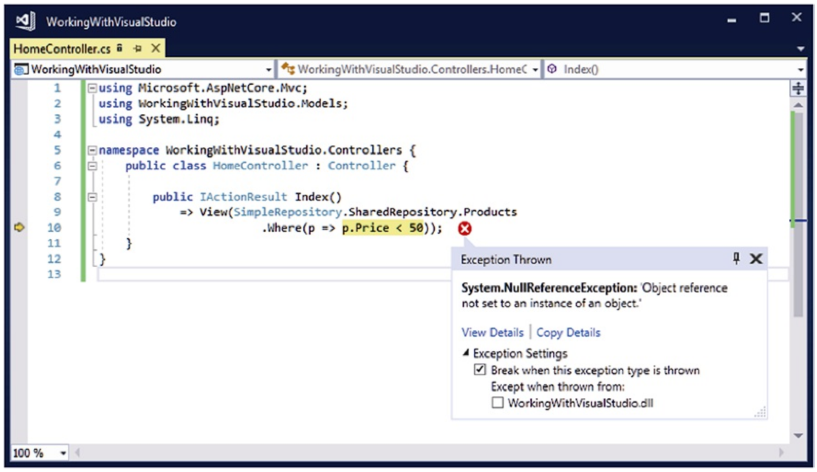


图6-11　未处理的异常

1）设置断点

调试器并不会指出导致问题的原因，只会指出问题出现的地方。Visual Studio高亮显示的语句表明在使用LINQ筛选对象时出了问题，但只需要做少量的工作就可以了解详细信息并找到根本原因。

断点（breakpoint）用于告诉调试器停止应用程序的执行而转为手动执行。可以检查应用程序的状态，查看正在发生的情况，还可以选择再次恢复执行。

要创建断点，请右击代码语句，然后从弹出的菜单中选择Breakpoint→Insert Breakpoint。作为演示，我们将断点应用于SimpleRepository类中的AddProduct方法，如图6-12所示。

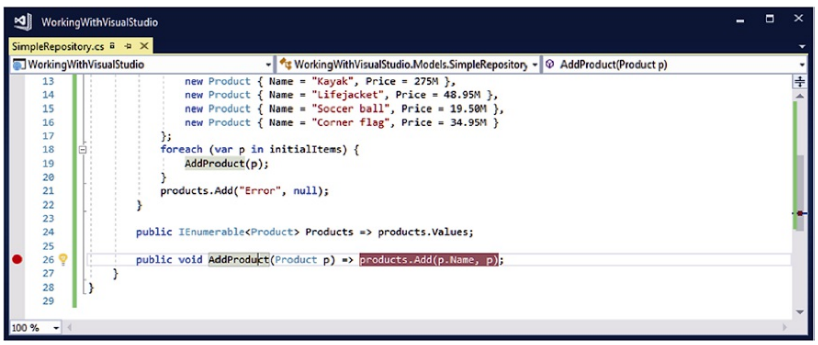


图6-12　创建断点

选择Debug→Start Debugging，启动应用程序，如果应用程序已经处于运行状态，就使用调试器或者选择Debug→Restart以重启应用程序。在浏览器初始化HTTP请求时，也会初始化SimpleRepository类。当程序执行到断点时，就会停止执行。

此时，可以使用Visual Studio的Debug菜单项或

窗口顶部的控件来控制应用程序的执行；或者选择Debug→Windows，启用调试视图来检查应用程序状态。

2）在代码编辑器中查看数据值

断点最常见的用途是在代码中跟踪bug。在修复bug之前，我们必须弄清楚究竟发生了什么，而Visual Studio提供的最有用的功能之一就是能够在代码编辑器中查看和监控变量的值。

如果将鼠标指针移动到变量p上，并移到调试器高亮显示的AddProduct方法上，就会弹出一个窗口，显示p的值，如图6-13所示。

这个例子可能不那么确切，因为数据对象与断点是在相同的构造函数中定义的，但这个功能适用于任何变量。可以浏览各个变量以查看其属性和字段值。每个变量的右侧都有一个小的图钉按钮，在

代码继续执行时可以用它监控变量的值。

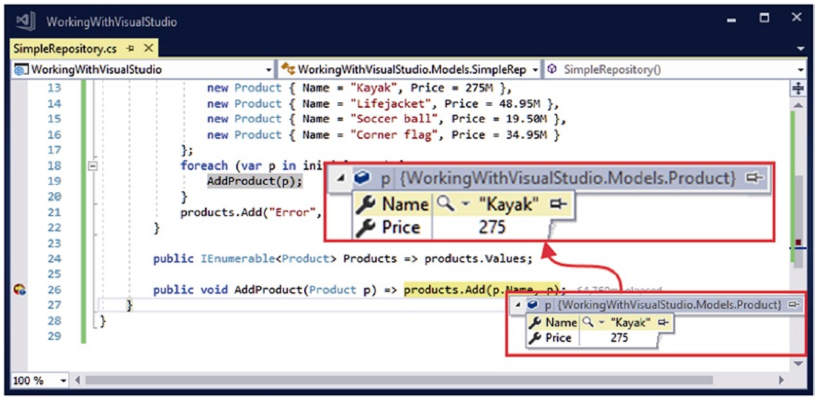


图6-13　检查数据值

将鼠标指针悬停在变量p上并固定到Product引用上，展开引用后，还可以锁定Name和Price属性，从而实现图6-14所示的效果。

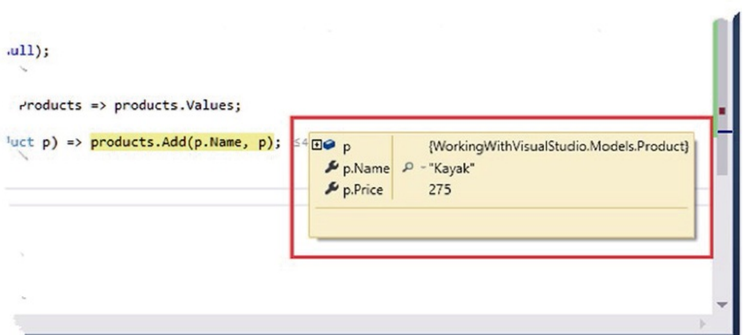


图6-14　在代码编辑器中锁定值

从Visual Studio的Debug菜单中选择Continue以继续执行应用程序。由于应用程序正在执行foreach循环，因此当再次遇到断点时将停止执行。被锁定的值将显示p变量及其属性值的变化，如图6-15所示。

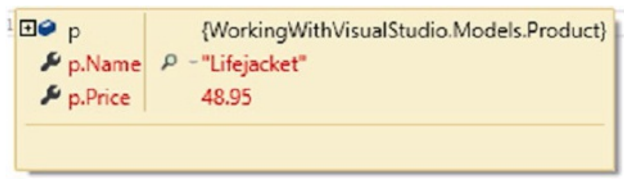


图6-15　监控锁定值的变化

3）使用Locals窗口

Locals窗口可通过选择Debug→Windows→Locals菜单项来打开。Locals窗口以类似于锁定值的方式显示数据值，但显示的是所有与断点相关的本地对象，如图6-16所示。

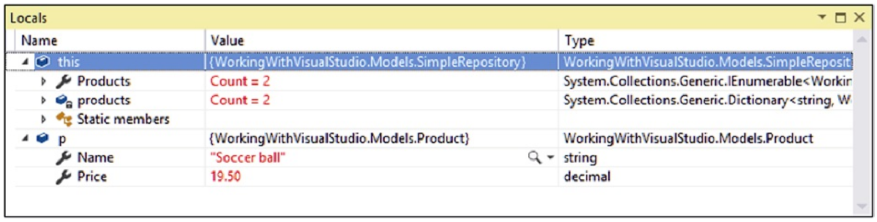


图6-16　Locals窗口

每次选择Debug→Continue时，应用程序就会继续执行，下一个对象将由foreach循环处理。如果一直继续下去，就会看到null引用出现，在Locals窗口中也可以看到。窗口和锁定的值显示在代码编辑器中。通过使用调试器控制应用程序的执行，可以跟踪代码的流程并了解正在发生的事情。

我们可以通过清理Product对象的集合来修复null引用问题，还可以选择另一种方法来使控制器更加健壮，如代码清单6-13所示，这里使用了null条件运算符来检查null值（如第4章所述）。

代码清单6-13　在HomeController.cs文件中解决null引用问题

```
using Microsoft.AspNetCore.Mvc;
```

```
using WorkingWithVisualStudio.Models;
using System.Linq;

namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {
        public IActionResult Index()
            => View(SimpleRepository.SharedRepository.P
roducts
                        .Where(p => p?.Price < 50));
    }
}
```

要禁用断点，可右击已应用断点的代码语句，并从弹出的菜单中选择Delete Breakpoint。重新启动应用程序，你将看到图6-17所示的简单数据表。

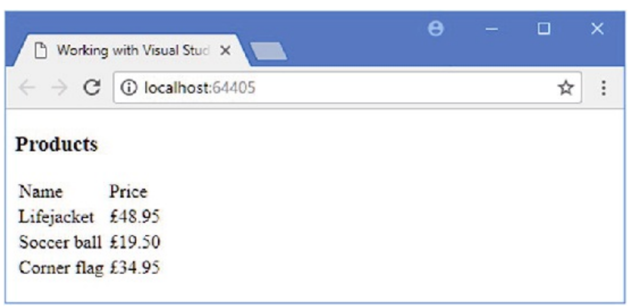


图6-17　修复错误

与那些需要使用错误扫描软件才能解决的问题相比，这个问题比较简单，但是Visual Studio调试

器是非常好用的，通过使用应用程序的各种可视化视图和控制执行操作，我们可以真正挖掘到错误的细节。

6.3.3 使用浏览器链接

浏览器链接（browser link）功能可以通过将一个或多个浏览器置于Visual Studio的控制之下来简化开发过程。尤其是在需要查看一系列浏览器的更改效果时，这项功能最有用。浏览器链接在使用类的自动编译功能时非常有效，因为它可以用来修改项目中的任何文件，并显示更改的效果，而不必切换到浏览器并手动重新加载页面。

1．设置浏览器链接

启用浏览器链接需要对Startup类进行配置更改，如代码清单6-14所示。

代码清单6-14　在WorkingWithVisualStudio文件夹下的Startup.cs文件中启用浏览器链接

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

2．使用浏览器链接

要了解浏览器链接的工作方式，请从Visual Studio的Debug菜单中选择Start Without Debugging。Visual Studio将启动应用程序并打开新的浏览器选项卡以显示结果。查看发送到浏览器的HTML，你将看到其中包含一块附加的内容，如下所示：

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
</head>
<body>
    <h3>Products</h3>
    <table>
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
                <tr><td>Lifejacket</td><td>£48.95</td><
/tr>
                <tr><td>Soccer ball</td><td>£19.50</td>
</tr>
                <tr><td>Corner flag</td><td>£34.95</td>
</tr>
        </tbody>
    </table>
```

```
<!-- Visual Studio Browser Link -->
<script type="application/json" id="__browserLink_initi
alizationData">
    {"requestId":"968949d8affc47c4a9c6326de21dfa03","re
questMappingFromServer":false}
</script>
<script type="text/javascript"
    src="http://localhost:55356/d1a038413c804e178ef009a
3be07b262/browserLink"
    async="async">
</script>
<!-- End Browser Link -->
</body>
</html>
```

提　示

如果没有看到附加部分，请从图6-18所示的菜单中选择Enable Browser Link并重新加载浏览器。

Visual Studio会将一对script元素添加到发送到浏览器的HTML中，以打开一个具有较长生命周期的HTTP连接并返回给应用程序服务器，以便Visual Studio可以要求浏览器重新加载页面。如果script元素不显示，请检查以确保在图6-18所示的菜单中选择了Enable Browser Link选项。代码清单6-15显示了对Index视图所做的更改，并演示了使用浏览器链接的效果。

代码清单6-15　向Index.cshtml文件添加一个时间戳

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Working with Visual Studio</title>
</head>
<body>
    <h3>Products</h3>
    <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")</p>
```

```
    <table>
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@($"{p.Price:C2}")</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

保存对视图文件所做的更改，并从Visual Studio工具栏的Browser Link菜单中选择Refresh Linked Browsers，如图6-18所示。如果浏览器链接不起作用，请尝试重新加载浏览器或重新启动Visual Studio并重试。

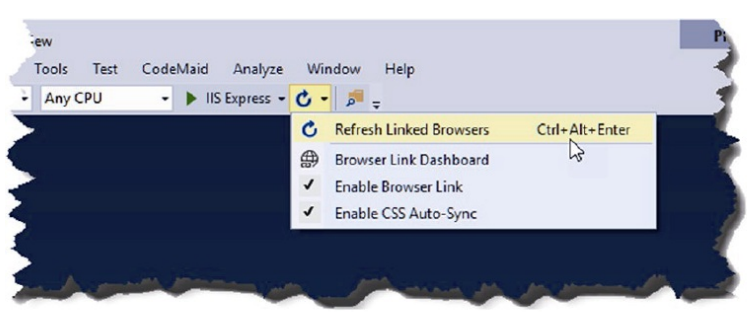


图6-18　使用浏览器链接重新加载浏览器

发送到浏览器的嵌入HTML中的JavaScript代码将重新加载页面，显示最终的效果——添加了一个简单的时间戳。每次选择Visual Studio菜单项时，浏览器都会对服务器发出一个新的请求。这个请求会使Index视图生成一个新的具有更新时间戳的HTML页面。

注　意

浏览器链接的script元素仅在成功的响应中才会嵌入，也就是说，如果在编译类文件、呈现Razor视图或处理请求时发生异常，浏览器和Visual Studio之间的连接就会丢失，并且一旦解决了问

题，就必须使用浏览器重新加载页面。

---

3. 使用多浏览器

浏览器链接可用于在多个浏览器中同时显示应用程序，这项功能在你希望解除浏览器之间的实现差异（特别是在实现自定义CSS样式表时）或查看应用程序如何在桌面浏览器和移动浏览器的组合中呈现时很有用。

要选取浏览器，请从Visual Studio工具栏的IIS Express下拉菜单中选择Browse With选项，如图6-19所示。

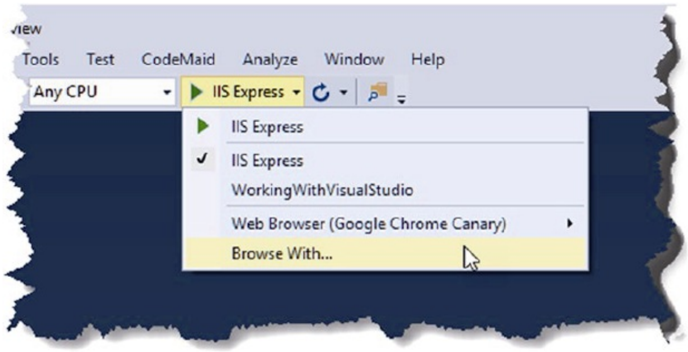


图6-19　选择多浏览器

Visual Studio会显示它所知道的浏览器的列表。图6-20显示了作者系统中安装的浏览器，其中一些是Windows操作系统自带的（比如Internet Explorer和Edge），另一些是作者自己安装的常用浏览器。

Visual Studio会在安装过程中查找常用的浏览器，你也可以使用Add按钮来设置未自动发现的浏览器。你还可以设置浏览器栈这样的第三方测试工具，通过将浏览器运行在云托管的虚拟机上，实现了不必人工管理大型的操作系统和浏览器矩阵就可以进行测试。

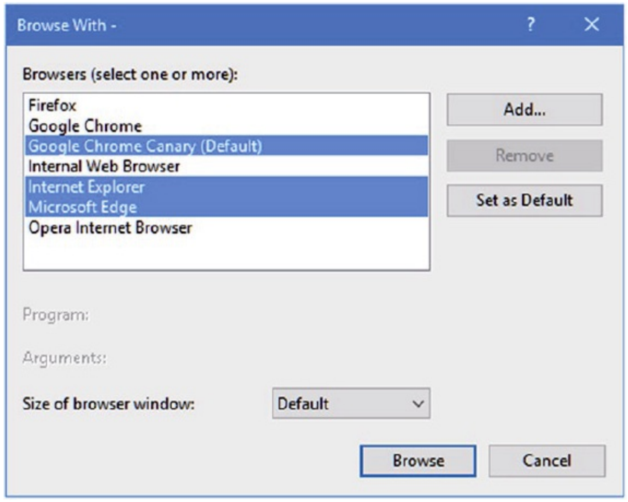


图6-20　从列表中选取浏览器

这里选择了Google Chrome Canary（Default）、Internet Explorer和Microsoft Edge浏览器。单击Browse按钮将启动这3种浏览器，并使它们加载示例应用程序的URL，如图6-21所示。

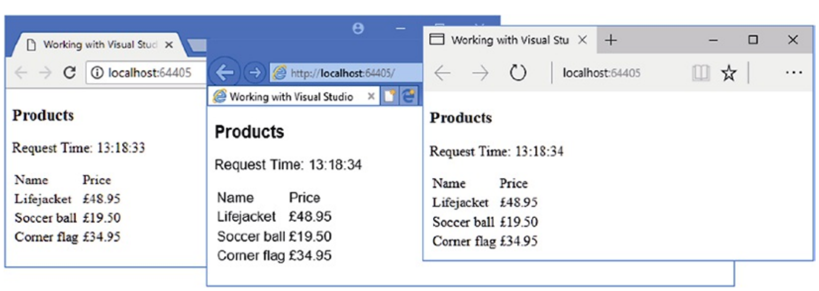


图6-21　使用多浏览器

可以通过选择Browser Link Dashboard菜单项来查看浏览器链接的管理情况，这将打开图6-22所示的窗口，其中显示了每个浏览器输出的是哪个URL的内容，每个浏览器都可以单独刷新。

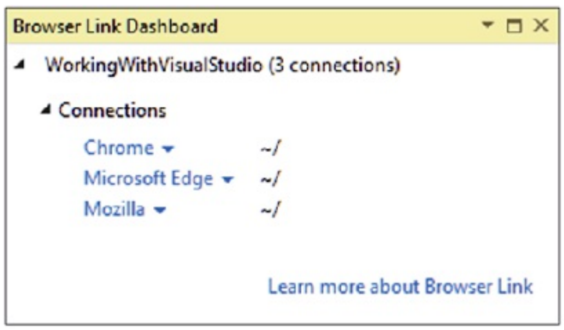


图6-22　Browser Link Dashboard窗口

6.4　部署JavaScript和CSS

在创建Web应用程序的客户端部分时，通常会创建许多自定义的JavaScript和CSS文件，它们用于补充由Bower安装的包。这些文件需要进行处理以优化它们在生产环境中的传递速率，从而最大限度地减少HTTP请求的数量以及将它们传送到客户端

所需的网络带宽。本节将介绍微软为执行这项任务而提供的VisualStudio扩展。

6.4.1　启用静态内容传递

ASP.NET Core允许从wwwroot文件夹向客户端提供静态文件，但是当使用Empty模板创建项目时，默认情况下这是不允许的。可通过将代码清单6-16所示的语句添加到Startup类中来启用对静态文件的支持。

代码清单6-16　在WorkingWithVisualStudio文件夹下的Startup.cs文件中启用对静态文件的支持

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

6.4.2　为项目添加静态内容

为了演示打包和缩小化处理过程，我们需要在项目中添加一些静态内容，并使其具备交付给客户端的能力。首先，创建wwwroot/css文件夹，这是存储自定义CSS文件的地方。然后，使用Style Sheet模板添加名为first.css的文件，如图6-23所示。Style Sheet模板位于ASP.NET Core→Web→Content部分。

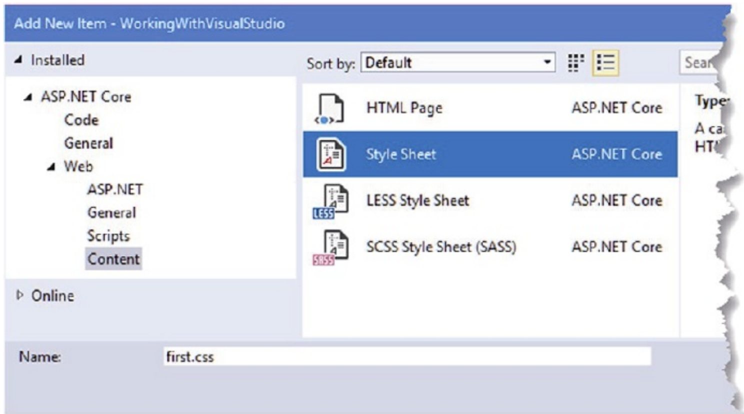


图6-23　添加first.css

编辑first.css文件以添加代码清单6-17所示的CSS样式。

代码清单6-17　wwwroot/css文件夹下的first.css文件的内容

```
h3 {
    font-size: 18pt;
    font-family: sans-serif;
}
table, td {
    border: 2px solid black;
    border-collapse:collapse;
    padding: 5px;
    font-family: sans-serif;
}
```

重复上述过程，在wwwroot/css文件夹中创建

另一个样式表，名为second.css，如代码清单6-18所示。

代码清单6-18　wwwroot/css文件夹下的second.css文件的内容

```
p {
    font-family: sans-serif;
    font-size: 10pt;
    color: darkgreen;
    background-color:antiquewhite;
    border: 1px solid black;
    padding: 2px;
}
```

自定义的JavaScript文件存储在wwwroot/js文件夹中。使用JavaScript File模板创建名为third.js的文件，如图6-24所示。

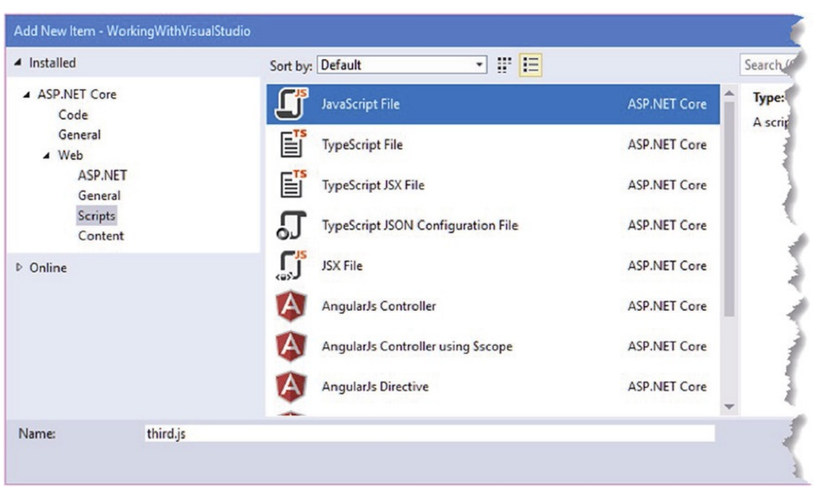


图6-24　创建一个JavaScript文件

在third.js文件中添加一些简单的JavaScript代码，如代码清单6-19所示。

代码清单6-19　wwwroot/js文件夹下的third.js文件的内容

```
document.addEventListener("DOMContentLoaded", function
() {
    var element = document.createElement("p");
    element.textContent = "This is the element from the
 third.js file";
    document.querySelector("body").appendChild(element)
;
});
```

这里还需要另外一个JavaScript文件。在wwwroot/js文件夹中创建fourth.js文件，在其中添加代码清单6-20所示的代码。

代码清单6-20　wwwroot/js文件夹下的fourth.js文件的内容

```
document.addEventListener("DOMContentLoaded", function
() {
    var element = document.createElement("p");
    element.textContent = "This is the element from the
```

```
 fourth.js file";
    document.querySelector("body").appendChild(element)
;
});
```

6.4.3　更新视图

最后的准备步骤是更新Index.cshtml文件以使用新的CSS样式表和JavaScript文件，如代码清单6-21所示。

代码清单6-21　为Views/Home文件夹下的Index.cshtml文件添加script和link元素

```
@model IEnumerable<WorkingWithVisualStudio.Models.Produ
ct>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
    <link rel="stylesheet" href="css/first.css" />
    <link rel="stylesheet" href="css/second.css" />
    <script src="js/third.js"></script>
    <script src="js/fourth.js"></script>
</head>
```

```
<body>
    <h3>Products</h3>
    <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")
</p>
    <table>
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@($"{p.Price:C2}")</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

如果运行示例应用程序，你将看到图6-25所示的内容。现有内容中已添加了CSS样式，JavaScript代码中也添加了新内容。

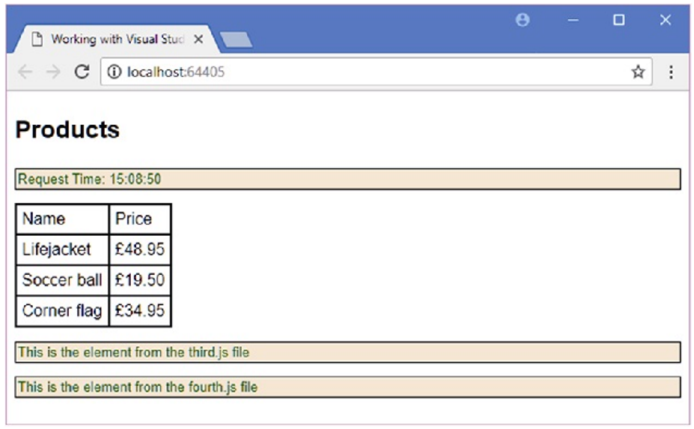


图6-25　运行示例应用程序

6.4.4　MVC应用程序中的打包和缩小

目前共有4个静态文件，所以浏览器必须发出4次请求才能获得这些静态文件，而且请求这些文件中的任何一个都比向客户端传递信息占用的带宽要大，因为它们包含了许多只对开发人员有意义但对浏览器没有意义的空白字符和变量名。

合并相同类型的文件称为打包。使文件变小称为缩小。这两个任务都是由Visual Studio的Bundle & Minifier扩展在ASP.NET Core MVC应用程序中执

行的。

1．下载Visual Studio扩展

第一步是安装扩展。选择Tools→Extensions and Updates菜单项，然后单击Online类别以显示可用的Visual Studio扩展库。在窗口右上角的搜索框中输入Bundler & Minifier，如图6-26所示。找到Bundler & Minifier扩展，然后单击Download按钮将其添加到Visual Studio中。完成安装并重新启动Visual Studio。

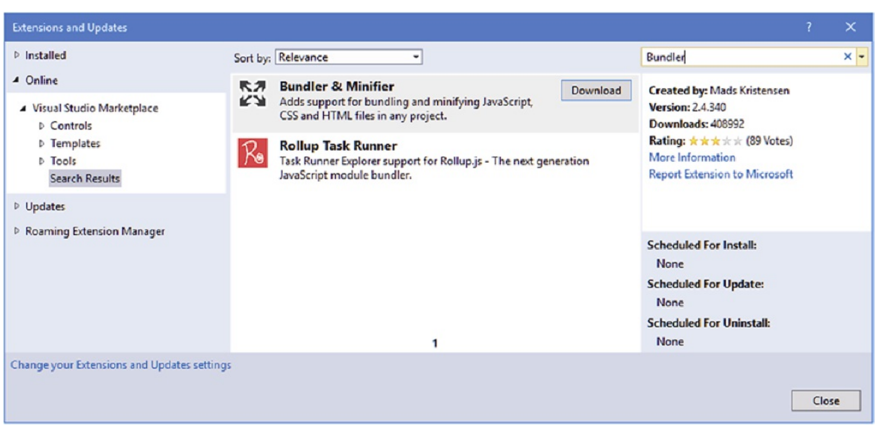


图6-26　找到Visual Studio扩展

2．打包和缩小文件

一旦安装Bundler & Minifier扩展并重新启动Visual Studio，就可以选择同一类型的多个文件，将它们打包在一起并缩小它们的内容。例如，在Solution Explorer窗格中选择first.css和second.css文件，右击后，从弹出的菜单中选择Bundler & Minifier→Bundle and Minify Files，如图6-27所示。

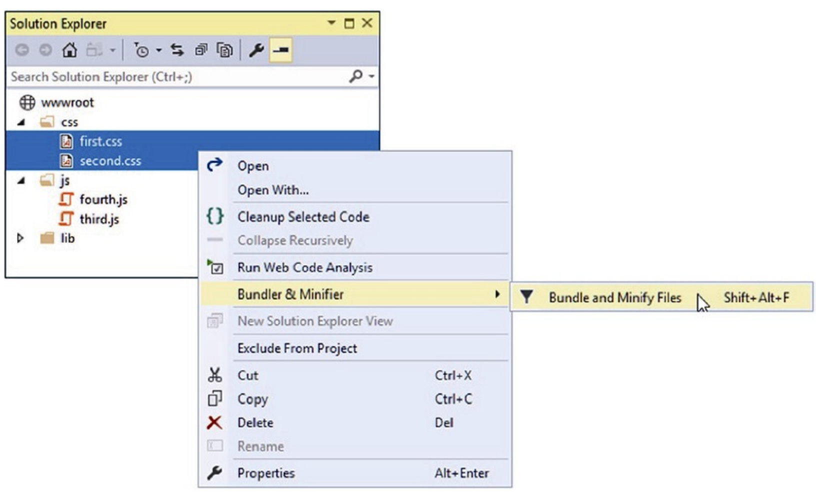


图6-27　打包和缩小CSS文件

将输出文件另存为bundle.css，Bundler & Minifier扩展将处理这个CSS文件。Solution Explorer

窗格中将显示新的bundle.css文件，可以展开它以显示bundle.min.css压缩文件。打开这个压缩文件后，你将看到两个独立的CSS文件的内容已被合并，并且所有的空白符都已被删除。你可能不愿意直接使用这个文件，但它非常小，只需要一个HTTP连接就能将CSS样式传递给客户端。

对third.js和fourth.js文件重复这个过程，在wwwroot/js文件夹中创建bundle.js和bundle.min.js文件。

警　告

一定要按照浏览器加载它们的顺序选择文件，以保留输出文件中的样式或代码语句的顺序。例

如，在选择fourth.js文件之前，请确保选择了third.js文件，以确保代码按正确的顺序执行。

在代码清单6-22中，已将单独文件的link元素替换为Index.cshtml文件中请求打包和缩小文件的元素。

代码清单6-22　在Index.cshtml文件中使用打包和压缩的文件

```
@model IEnumerable<WorkingWithVisualStudio.Models.Produ
ct>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
    <link rel="stylesheet" href="css/bundle.min.css" />
    <script src="js/bundle.min.js"></script>
</head>
<body>
    <h3>Products</h3>
    <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")
```

```
</p>
    <table>
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@($"{p.Price:C2}")</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

运行应用程序，虽然不会有任何可视化的更改，但打包和压缩的文件已经向浏览器提供了在单独的文件中定义的所有样式和代码。

在执行打包和缩小操作时，Bundler & Minifier 扩展会将文件的处理记录在bundleconfig.json文件中并保存在项目的根目录下。下面是为示例应用程序中的文件生成的配置：

```
[
```

```
  {
    "outputFileName": "wwwroot/css/bundle.css",
    "inputFiles": [
      "wwwroot/css/first.css",
      "wwwroot/css/second.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/bundle.js",
    "inputFiles": [
      "wwwroot/js/third.js",
      "wwwroot/js/fourth.js"
    ]
  }
]
```

Bundler & Minifier扩展将自动监控输入文件的更改，并在发生更改时重新生成输出文件，以保证所做的任何编辑都反映在打包和缩小的文件中。作为演示，代码清单6-23显示了对third.js文件所做的更改。

代码清单6-23　third.js文件中的改动

```
document.addEventListener("DOMContentLoaded", function
() {
    var element = document.createElement("p");
    element.textContent = "This is the element from the
```

```
 (modified) third.js file";
    document.querySelector("body").appendChild(element)
;
});
```

保存文件后，Bundler & Minifier扩展将重新生成bundle.min.js文件。如果重新加载浏览器，你将看到图6-28所示的更改效果。

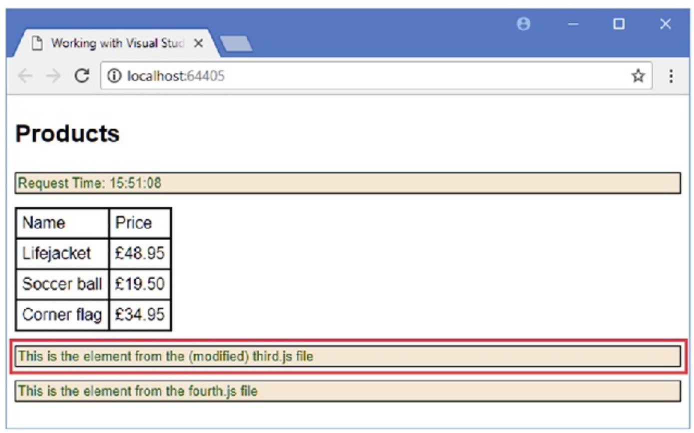


图6-28　打包和缩小的文件中的更改检测

6.5　小结

本章介绍了MVC项目的结构，描述了两个可用的.NET运行时功能，以及Visual Studio为Web应用开发提供的一些功能，包括自动类编译、浏览器

链接以及打包和缩小。下一章将介绍ASP.NET Core MVC项目是如何进行单元测试的。

第7章　对MVC应用程序进行单元测试

本章演示如何对MVC应用程序进行单元测试。单元测试是一种测试形式，由于其中的单个组件已与应用程序的其他部分隔离，因此可以彻底验证其行为。ASP.NET Core MVC已设计为便于创建单元测试，Visual Studio为各种单元测试框架提供了支持。本章展示如何设置单元测试项目，说明如何安装最流行的测试框架之一，并描述编写和运行测试的过程。表7-1总结了本章要完成的操作。

表**7-1**　本章要完成的操作

操作	方法	代码清单
创建单元测试	创建单元测试项目，安装一个测试包，添加包含测试的类	代码清单7-5和代码清单7-6

为单元测试隔离组件	使用接口分离应用程序组件，在单元测试中使用带有受限测试数据的伪实现	代码清单7-7～代码清单7-14
对不同的数据值运行同样的单元测试	使用参数化的单元测试，也可通过方法或属性获取测试数据	代码清单7-15～代码清单7-17
创建伪测试对象	使用mocking框架	代码清单7-18和代码清单7-19

决定是否进行单元测试

能够简单地进行单元测试是使用ASP.NET Core MVC的优势之一，但这不适用于所有人。

作者喜欢单元测试，经常在自己的项目中使用单元测试，但不是所有的项目，并且也不像你所预期的那样。作者会专门为特性和功能编写单元测试，它们难以编写，而且很可能是部署中bug的来源。在这种情况下，单元测试有助于确立那些重要的想法。仅仅测试需要实现的东西，可以帮助处理

关于潜在问题的想法，这应该发生在开始处理实际的bug之前。

也就是说，单元测试是一种工具，只有你才知道需要执行多少测试。如果没有发现单元测试的好处，或者有更适合的方法论，那就不要因为单元测试时髦而使用它（但是，如果没有更好的方法论，并且根本没有执行测试，那么你可能会让用户发现bug，这非常不理想。你不必做单元测试，但是你真的需要考虑做一些其他测试）。

如果以前没有接触过单元测试，那么鼓励你尝试一下，看看它是如何工作的。如果你不是单元测试的粉丝，那么可以跳过本章，继续进入第8章，在那里构建更现实的MVC应用程序。

7.1　准备示例项目

本章将继续使用你在第6章中创建的项目。对于本章，将在仓储中添加新的Product对象以添加支持。

7.1.1　启用内置的标签助手

本章使用内置的标签助手为链接元素设置href属性。本书将在第23～25章说明标签助手是如何工作的。为了简单地启用它们，可右击Views文件夹，从弹出的菜单中选择Add New Item，然后从ASP.NET类别中选择MVC View Imports Page项目模板，创建一个视图导入文件，Visual Studio将自动设置文件名为_ViewImports.cshtml，然后单击Add按钮，添加代码清单7-1所示的代码。

代码清单7-1　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

以上语句将启用内置的标签助手，包括你很快要在Index视图中使用的标签助手。可以添加using语句，从项目中导入命名空间，但是Index视图不是本章中示例应用程序的重要部分，并且使用命名空间引用模型类型也不是问题。

7.1.2　为控制器添加操作方法

第一步是为Home控制器添加操作方法，Home控制器用来渲染通过浏览器输入和接收数据的视图，如代码清单7-2所示。这些操作方法遵循你第2章使用过的模式，第17章将详细说明这些模式。

代码清单7-2　在Controller文件夹下的HomeController.cs文件中添加操作方法

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;
namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {
```

```
        SimpleRepository Repository = SimpleRepository.
SharedRepository;

        public IActionResult Index() => View(Repository
.Products
                    .Where(p => p?.Price < 50));

        [HttpGet]
        public IActionResult AddProduct() => View(new P
roduct());

        [HttpPost]
        public IActionResult AddProduct(Product p) {
            Repository.AddProduct(p);
            return RedirectToAction("Index");
        }
    }
}
```

7.1.3　创建数据输入表单

为允许用户创建新的产品，在Views/Home文件夹中创建名为AddProduct.cshtml的Razor视图，并且与Home控制器中的AddProduct方法所呈现的视图一样，遵循默认的文件名和位置约定。代码清单7-3展示了新视图的内容，它基于你在第6章使用

Bower加入项目的Bootstrap CSS包。

代码清单7-3　Views/Home文件夹下的AddProduct.cshtml文件的内容

```
@model WorkingWithVisualStudio.Models.Product
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.min.css" />
</head>
<body class="p-2">
    <h3 class="text-center">Create Product</h3>
    <form asp-action="AddProduct" method="post">
        <div class="form-group">
            <label asp-for="Name">Name:</label>
            <input asp-for="Name" class="form-control"
/>
        </div>
        <div class="form-group">
            <label asp-for="Price">Price:</label>
            <input asp-for="Price" class="form-control"
 />
        </div>
        <div class="text-center">
            <button type="submit" class="btn btn-primar
```

```
y">Add</button>
            <a asp-action="Index" class="btn btn-second
ary">Cancel</a>
        </div>
    </form>
</body>
</html>
```

该视图包含一个HTML表单，其使用HTTP POST请求发送Name和Price值到Home控制器的AddProduct操作方法。内容已使用Bootstrap CSS包进行装饰。

7.1.4　更新Index视图

最后的准备步骤是更新Index视图，以便包含到新表单的链接，如代码清单7-4所示。这里还借此机会删除了你在上一章中使用的JavaScript文件，使用Bootstrap替换了自定义的CSS样式表，并应用于视图中的HTML元素。

代码清单7-4　更新Views/Home文件夹下的Index.cshtml文件的

```
@model IEnumerable<WorkingWithVisualStudio.Models.Produ
ct>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Working with Visual Studio</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.min.css" />
</head>
<body class="p-1">
    <h3 class="text-center">Products</h3>
    <table class="table table-bordered table-striped">
        <thead>
            <tr><td>Name</td><td>Price</td></tr>
        </thead>
        <tbody>
            @foreach (var p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>@($"{p.Price:C2}")</td>
                </tr>
            }
        </tbody>
    </table>
    <div class="text-center">
        <a class="btn btn-primary" asp-action="AddProdu
ct">
            Add New Product
```

```
        </a>
    </div>
</body>
</html>
```

如果运行示例应用程序，你将会看到新装饰的内容和Add New Product按钮，用于引导到数据输入表单。如果提交表单，将添加新的Product对象到存储库中，并重定向浏览器以便初始的应用程序视图被展示出来，如图7-1所示。

提　示

切记此例中的存储库仅仅在内存中存储对象，这意味着在应用程序重新启动之时，创建的任何新产品都将被丢弃。

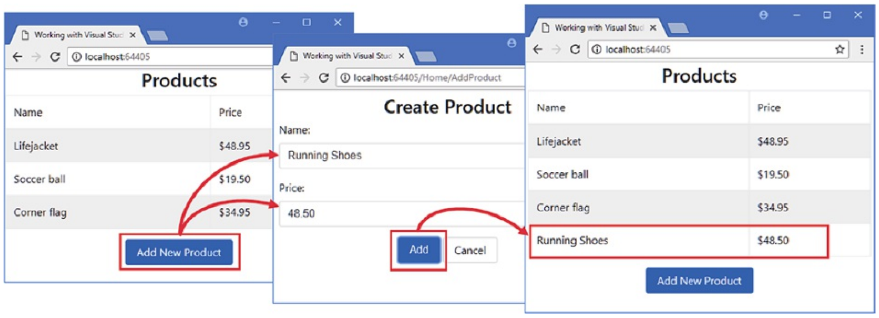


图7-1　运行示例应用程序

7.2　测试MVC应用程序

单元测试用于验证应用程序中独立组件的行为和特性，ASP.NET Core和ASP.NET Core MVC被设计为尽可能为Web应用程序设置和运行单元测试。本节将说明如何在Visual Studio中设置单元测试，并演示如何为MVC应用程序编写单元测试，还将介绍一些使得单元测试更为简单和可靠的有用工具。

有多种单元测试包可用，本书中使用的名为xUnit.net，它与Visual Studio能够良好集成，可用来

为ASP.NET编写单元测试。表7-2介绍了xUnit.net的背景。

表**7-2　xUnit.net**的背景

问题	答案
xUnit.net是什么？	xUnit.net是单元测试框架，可以用来测试ASP.NET Core MVC应用程序
为何有用？	xUnit.net是编写良好的测试框架，可以很容易集成到Visual Studio中
如何使用？	测试被定义为带有Fact或Theory特性注解的方法。在方法体中，使用Assert类定义的方法对测试期望的结果与实际的值进行比较
有何缺陷或限制？	单元测试的主要缺陷是不能有效隔离测试中的组件。xUnit.net特有的最大问题是缺失文档。一些基本信息请参见GitHub网站，但是高级用法需要做一些试验
有何替代品？	有许多测试框架可用，两个流行的选择是MSTest（来自微软）和NUnit

注　意

单元测试中的任何问题都与个人偏好有关，并且每个人的意见并不相同。类似地有些开发者不喜欢将单元测试的代码从应用程序代码中分离出来，倾向于将测试定义在同一个项目中，甚至定义在同一个类文件中。这里介绍的是常用方式，也是本书遵循的方式，但是，如果不适合你，那么你应该尝试其他的测试风格，直至找到自己喜欢的。

7.2.1　创建单元测试项目

对于ASP.NET Core应用程序，通常创建单独的Visual Studio项目来保存单元测试，每个测试定义为C#类中的一个方法。使用单独项目意味着可以部

署应用程序而不需要部署测试。

为了创建测试项目，在Solution Explorer窗格中，右击WorkingWithVisualStudio解决方案，从弹出的菜单中选择Add→New Project。在弹出的Add New Project对话框中，在左侧选择Visual C#→.NET Core，在中间选择xUnit Test Project (.NET Core)模板，如图7-2所示。

警　告

确保选择了正确的项目模板。Visual Studio 提供了多种单元测试的项目模板，并且它们有类似的名称。

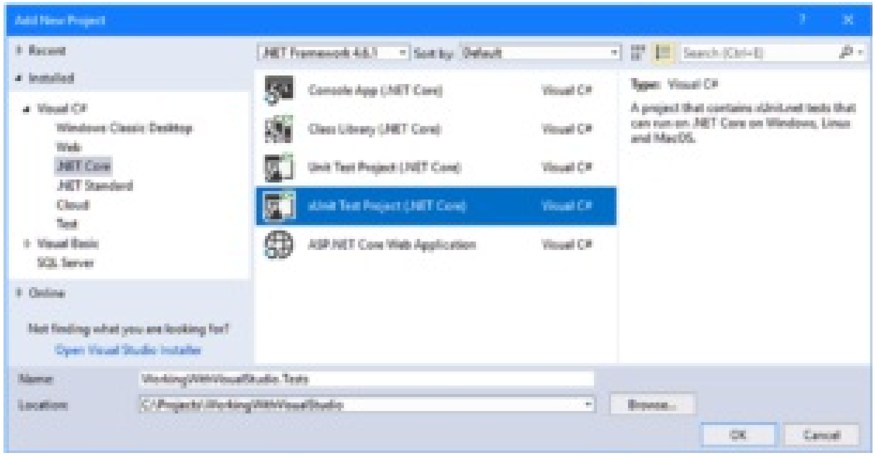


图7-2　选择xUnit Test Project (.NET Core)模板

约定的命名方式是将单元测试项目命名为<ApplicationName>.Tests。将新项目的名称设置为Working WithVisualStudio.Tests，然后单击OK按钮以创建新项目。Visual Studio将会创建项目并安装xUnit对应的 NuGet 包以及依赖的包。

删除默认测试类

Visual Studio添加了一个C#类到该测试项目中，它会导致后面示例的测试结果变得难以理解。

右击WorkingWithVisualStudio.Tests项目中的UnitTest1.cs文件，然后从弹出菜单中选择Delete。在提示对话框中单击OK按钮，Visual Studio将会删除该类文件。

7.2.2　创建项目引用

为了使主项目中的类对于测试项目可用，右击Solution Explorer窗格中的WorkingWithVisualStudio.Tests项，然后从弹出菜单中选择Add→Reference。选中Solution部分的WorkingWithVisualStudio项，如图7-3所示。

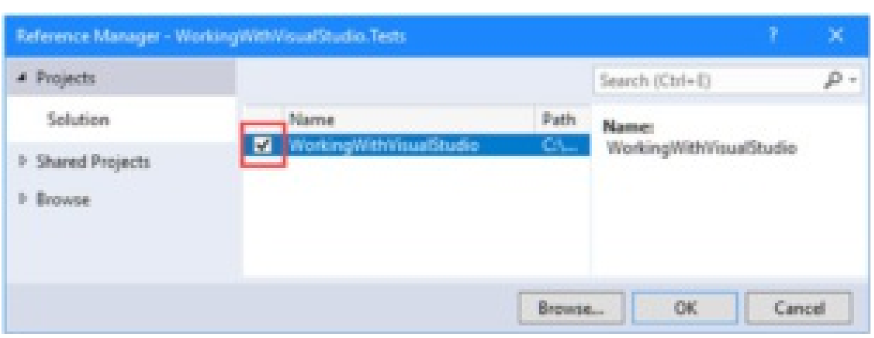


图7-3　选择WorkingWithVisualStudio项

单击OK按钮以创建对应用程序项目的引用。你可能在Solution Explorer窗格的测试项目的Dependencies中看到一个处理中的图标，但是一旦你构建该项目这个图标将会消失。

编写和运行单元测试

现在所有的准备工作已经完成，可以编写一些测试。为了开始，如代码清单7-5所示，在WorkingWith- VisualStudio.Tests项目中添加名为ProductTests.cs的类文件，这是一个简单的类文件，但是其中包含了从单元测试开始所需的一切。

代码清单7-5　WorkingWithVisualStudio.Tests文件夹下的ProductTests.cs文件的内容

```
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {

    public class ProductTests {
```

```
        [Fact]
        public void CanChangeProductName() {

            // Arrange
            var p = new Product { Name = "Test", Price
= 100M };

            // Act
            p.Name = "New Name";

            //Assert
            Assert.Equal("New Name", p.Name);
        }

        [Fact]
        public void CanChangeProductPrice() {

            // Arrange
            var p = new Product { Name = "Test", Price
= 100M };

            // Act
            p.Price = 200M;

            //Assert
            Assert.Equal(100M, p.Price);
        }
    }
}
```

注　意

CanChangeProductPrice方法包含故意设置的错误，这将在本节稍后部分解决。

ProductTests类中有两个单元测试，其中的每个来自WorkingWithVisualStudio项目中Product模型类的不同行为。测试项目包含许多类，其中的每个可以包含多个单元测试。

通常，测试方法的名称描述了测试的内容，类的名称描述了正在测试的内容。这使得在项目中容易组织测试，在通过Visual Studio运行测试的时候，容易理解所有测试的结果。例如，名称ProductTests指出测试的是Product对象。

应用于每个方法的Fact特性指出这是一个测试。在方法体中，单元测试遵循称为组织（Arrange）、行动（Act）、断言（Assert）的A/A/A模式。组织是指为测试设置条件，行动是指执行测试，断言是指验证结果是否像预期的那样。

测试的组织和行动部分都是常规的C#代码，但是断言部分由xUnit.net处理，xUnit.net提供了一个名为Assert的类，其方法用于检查行动的结果是否如预期的那样。

提　示

Fact特性和Assert类都定义在命名空间Xunit中，因此所有的测试类中都必须有using语句。

Assert类的方法是静态的，用于比较不同种类的期望值和实际值，表 7-3 展示了 Assert 方法的常见用法。

表**7-3　xUnit.net**提供的**Assert**方法的常见用法

名称	描述
Equal(expected,result)	断言结果与期望值相等。该方法的重载版本用于比较不同的类型和集合。该方法的另一个版本接收一个额外的参数对象，该对象实现了用于比较对象的IEqualityComparer<T>接口
NotEqual(expected,result)	断言结果不等于期望值
True(result)	断言结果为true
False(result)	断言结果为false
IsType(expected,result)	断言结果为指定类型
IsNotType(expected,result)	断言结果不为指定类型
IsNull(result)	断言结果为null

IsNotNull(result)	断言结果不为null
InRange(result,low,high)	断言结果介于low和high
NotInRange(result,low,high)	断言结果不介于low和high
Throws(exception,expression)	断言指定的表达式抛出指定的异常类型

每个Assert方法都允许进行不同类型的比较，如果得不到预期的结果，就引发异常。异常用于标志测试失败。在代码清单7-5所示的测试中，使用Equal方法来检测属性的值是否被正确修改：

```
...
Assert.Equal("New Name", p.Name);
...
```

7.2.3　编写并运行单元测试

Visual Studio支持通过测试管理器（Test Explorer）查找和运行测试。可以通过选择

Test→Windows→Test Explorer找到测试管理器，如图7-4所示。

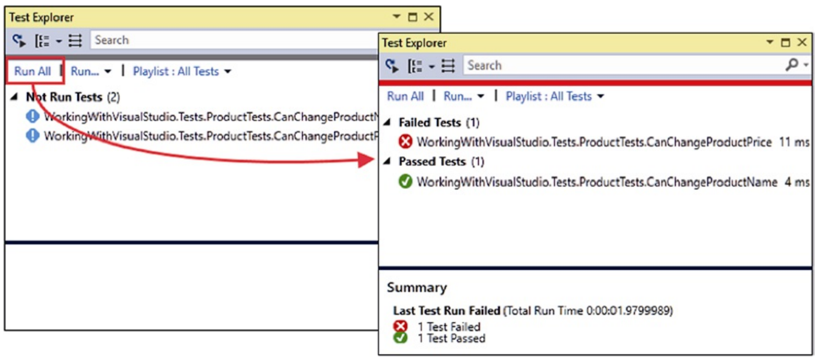


图7-4　Visual Studio的测试管理器

提　示

如果在测试管理器中没有看到单元测试，就请重新构建解决方案，编译触发单元测试的发现过程。

可通过单击Test Explorer窗口中的Run All按钮来运行全部测试。Visual Studio将使用xUnit.net来运行项目中的测试并显示结果。如上所述，CanChangeProductPrice测试包含导致测试失败的错误。问题源自AssertEqual方法的参数，其使用测试结果来比较源Price属性值而不是修改的值。代码清单7-6纠正了这个错误。

代码清单7-6　更正ProductTests.cs文件中的测试

```
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {

    public class ProductTests {

        [Fact]
        public void CanChangeProductName() {
            // Arrange
            var p = new Product { Name = "Test", Price
= 100M };
```

```
            // Act
            p.Name = "New Name";

            //Assert
            Assert.Equal("New Name", p.Name);
        }

        [Fact]
        public void CanChangeProductPrice() {

            // Arrange
            var p = new Product { Name = "Test", Price
= 100M };

            // Act
            p.Price = 200M;

            //Assert
            Assert.Equal(200M, p.Price);
        }
    }
}
```

提　示

当测试失败时，最好先检查测试的准确性，再检查测试的组件，特别是测试新的或刚刚修改的组件。

如果拥有大量测试，那么全部运行测试需要花费一些时间。为了快速迭代，测试管理器提供了不同的选项来测试子集。最有用的子集是失败的测试集，如图7-5所示。重新运行不正确的测试，测试管理器将显示没有测试失败。

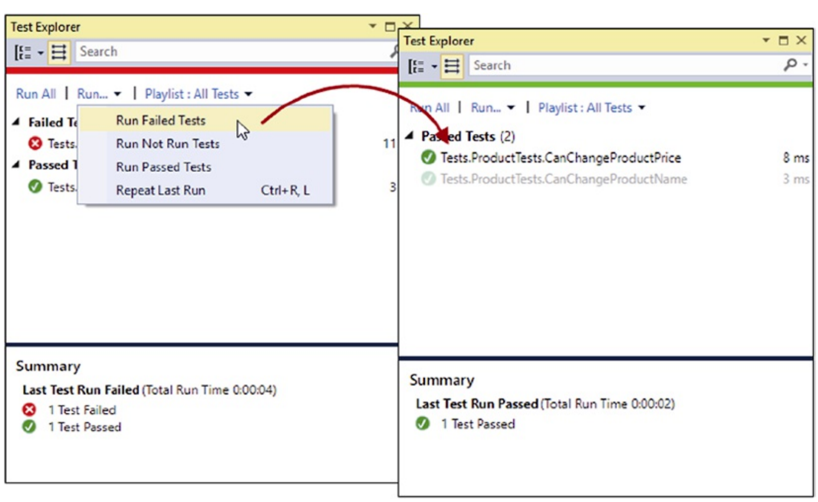


图7-5　有选择地运行测试

7.2.4　隔离组件以进行单元测试

为类似于Product这样的类编写单元测试是很容易的。不是因为Product类简单，而是因为Product类是自包含的，这意味着当在Product对象上执行操作时，可以确认是在测试由Product类提供的功能。

对于MVC应用程序中的其他组件来说，情况会变得复杂，因为它们之间存在依赖。你所定义的下一个测试集将处理控制器，检查在控制器和视图之间传递的Product对象的顺序。

当比较通过自定义的类实例化的对象时，需要使用xUnit.net中的Assert.Equal方法来接收一个实现了IEqualityComparer<T>接口的参数，以便对象可以比较。第一步是在测试项目中添加名为Comparer.cs的文件，用它定义代码清单7-7中的类。

代码清单7-7　WorkingWithVisualStudio.Tests项目中Comparer.cs文件的内容

```
using System;
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Tests {

    public class Comparer {

        public static Comparer<U> Get<U>(Func<U, U, boo
l> func) {
            return new Comparer<U>(func);
        }
    }

    public class Comparer<T> : Comparer, IEqualityCompa
rer<T> {
        private Func<T, T, bool> comparisonFunction;

        public Comparer(Func<T, T, bool> func) {
            comparisonFunction = func;
        }

        public bool Equals(T x, T y) {
            return comparisonFunction(x, y);
        }

        public int GetHashCode(T obj) {
            return obj.GetHashCode();
        }
    }
}
```

这些类允许使用Lambda表达式，而不是针对每种组件类型定义新的类以创建IEqualityComparer<T>对象。这不是必需的，但能简化单元测试类中的代码，使其易读、易于维护。

现在很容易进行比较，可以说明应用程序组件之间的依赖问题。添加名为HomeControllerTests.cs的文件到WorkingWithVisualStudio.Tests项目中，定义代码清单7-8所示的单元测试。

代码清单7-8　WorkingWithVisualStudio.Tests项目中HomeControllerTests.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        [Fact]
        public void IndexActionModelIsComplete() {
            // Arrange
```

```
            var controller = new HomeController();

            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(SimpleRepository.SharedReposit
ory.Products, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }
    }
}
```

以上代码清单中的单元测试会检查Index操作方法传递给视图的所有存储库中的对象（此刻忽略行动部分，第17章将说明ViewResult类及其在MVC应用程序中扮演的角色）。目前，知道Index操作方法返回的模型数据就足够了。

如果运行测试，你将看到测试失败了，指出存储库中的对象集不同于Index操作方法返回的对象集。为了弄清楚失败的原因，有个问题需要解决：

假设测试是在Home控制器上执行，但是控制器类依赖于SimpleRepository类，这使得难以揭示错误来自测试目标还是应用程序的另一部分。

这个示例应用程序简单到可以通过查看HomeController和SimpleRepository类的代码，就能轻松指出问题所在。在实际应用程序中，目视检查并不容易，依赖链使得代码难以理解导致测试失败。通常，存储库将依赖于某种持久化存储系统（比如数据库）以及提供访问的库，单元测试可作用于整个复杂的组件链，其中任何一个都可能导致错误。

当目标只针对应用程序的一小部分时，单元测试是有效的，比如独立的方法或类。你需要做的是能够从应用程序的其他部分隔离Home控制器，以便可以限制测试的范围，并排除任何存储库导致的影响。

隔离组件

隔离组件的关键是使用C#接口。为了将控制器和存储库分离，在Models文件夹中添加名为IRepository.cs的类文件，用它定义代码清单7-9所示的接口。

代码清单7-9　Models文件夹下的IRepository.cs文件的内容

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {

    public interface IRepository {

        IEnumerable<Product> Products { get; }
        void AddProduct(Product p);
    }
}
```

这个接口没有什么特别之处（除没有定义常见Web应用程序需要的全部操作之外。但是，添加这样的接口可以使你能够轻松隔离组件进行测试。第一步是更新SimpleRepository类以便实现新的接口，

如代码清单7-10所示。

代码清单7-10　在Model文件夹下的SimpleRepository.cs文件中实现接口

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
    public class SimpleRepository : IRepository {
        private static SimpleRepository sharedRepositor
y = new SimpleRepository();
        private Dictionary<string, Product> products
            = new Dictionary<string, Product>();

        public static SimpleRepository SharedRepository
 => sharedRepository;

        public SimpleRepository() {
            var initialItems = new[] {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M },
                new Product { Name = "Corner flag", Pri
ce = 34.95M }
            };
            foreach (var p in initialItems) {
                AddProduct(p);
            }
            products.Add("Error", null);
```

```
        }

        public IEnumerable<Product> Products => product
s.Values;

        public void AddProduct(Product p) => products.A
dd(p.Name, p);
    }
}
```

下一步是修改控制器，以便引用存储库的属性使用接口而不是类，如代码清单7-11所示。

代码清单7-11　在Controller文件夹下的HomeController.cs文件中添加存储库属性

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;

namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {
        public IRepository Repository = SimpleRepositor
y.SharedRepository;

        public IActionResult Index() => View(Repository
.Products
                    .Where(p => p?.Price < 50));
        [HttpGet]
        public IActionResult AddProduct() => View();
```

```
        [HttpPost]
        public IActionResult AddProduct(Product p) {
            Repository.AddProduct(p);
            return RedirectToAction("Index");
        }
    }
}
```

提　示

ASP.NET Core MVC支持使用更优雅的方式来解决这个问题，这称为依赖注入，将在第18章说明。依赖注入经常导致混乱，因此本章将以更简单、更手动的方式来隔离组件。

这可能不是一个重大变化，但是它允许在测试期间改变控制器使用的存储库。代码清单7-12更新

了控制器的单元测试，以便它们使用特殊版本的存储库。

代码清单7-12　在HomeControllerTests.cs文件的单元测试中隔离控制器

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        class ModelCompleteFakeRepository : IRepository
 {

            public IEnumerable<Product> Products { get;
 } = new Product[] {
                new Product { Name = "P1", Price = 275M
 },
                new Product { Name = "P2", Price = 48.9
5M },
                new Product { Name = "P3", Price = 19.5
0M },
                new Product { Name = "P3", Price = 34.9
5M }};
            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
```

```
        }

        [Fact]
        public void IndexActionModelIsComplete() {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepository();

            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }
    }
}
```

这里定义了一个假的IRepository接口实现，它仅仅实现了测试需要的属性，并使用始终一致的测试数据（使用真实数据库可能不是这种情况，特别是在你与其他开发人员共享，并且他们正在进行自己的修改时）。

修订之后的单元测试还是失败了，表明问题来自HomeController类的Index操作方法，而不是依赖的组件。通过单元测试进行的操作方法是非常简单的，检查问题也是显而易见的。

```
...
public IActionResult Index() => View(Repository.Product
s.Where(p => p.Price < 50));
...
```

问题来自LINQ表达式的Where方法，Where方法用于过滤Price属性大于或等于50的Product对象。在这一点上，可以确信产生问题的原因。但是在做修改之前，最佳实践是创建测试来确认问题，如代码清单7-13所示。

代码清单7-13　在WorkingWithVisualStudio.Tests项目的HomeControllerTests.cs文件中添加测试

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
```

```
using Xunit;
namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        class ModelCompleteFakeRepository : IRepository
 {

            public IEnumerable<Product> Products { get;
 } = new Product[] {
                new Product { Name = "P1", Price = 275M
 },
                new Product { Name = "P2", Price = 48.9
5M },
                new Product { Name = "P3", Price = 19.5
0M },
                new Product { Name = "P3", Price = 34.9
5M }};

            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
        }

        [Fact]
        public void IndexActionModelIsComplete() {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepository();

            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
```

```
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }

        class ModelCompleteFakeRepositoryPricesUnder50
: IRepository {

            public IEnumerable<Product> Products { get;
 } = new Product[] {
                new Product { Name = "P1", Price = 5M },

                new Product { Name = "P2", Price = 48.9
5M },
                new Product { Name = "P3", Price = 19.5
0M },
                new Product { Name = "P3", Price = 34.9
5M }};

            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
        }

        [Fact]
        public void IndexActionModelIsCompletePricesUnd
er50() {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepositoryPricesUnder50();

            // Act
```

```
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }
    }
}
```

提　示

这些测试中含有大量的重复代码。7.3节将描述如何简化测试。

上面定义了一个新的仅含有Price属性值小于50

的Product对象的存储库，并且你将在新的测试中仅使用这个存储库。如果运行这个测试，你将看到测试成功了，这增加了问题是由Index操作方法中Where方法的用法导致的权重。

在真实项目中，了解测试失败的原因时，需要将测试的目的与应用程序的规范相协调。可能的情况是，Index操作方法应该通过Price来过滤Product对象，在这种情况下，将需要修改测试。这是常见的结果，失败的测试并不总是表明应用程序中真正存在问题。从另一个角度说，如果Index操作方法不应该过滤模型对象，则需要更正修改，如代码清单7-14所示。

代码清单7-14　移除Controller文件夹下的HomeController.cs文件中的LINQ过滤器

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;

```

```
namespace WorkingWithVisualStudio.Controllers {
    public class HomeController : Controller {
        public IRepository Repository = SimpleRepositor
y.SharedRepository;

        public IActionResult Index() => View(Repository
.Products);

        [HttpGet]
        public IActionResult AddProduct() => View(new P
roduct());

        [HttpPost]
        public IActionResult AddProduct(Product p) {
            Repository.AddProduct(p);
            return RedirectToAction("Index");
        }
    }
}
```

测试驱动开发

本章遵循最常见的单元测试风格，先开发应用程序的功能，再进行测试以确保按需工作。因为大多数开发者首先考虑应用程序代码，然后进行测

试，所以这种风格很受欢迎。

这种开发方式的问题在于倾向于只关注部分应用程序代码，这些代码要么难以编写，要么需要重度调试，从而使功能的某些方面仅仅得到部分测试或未经测试。

另一种开发方式是测试驱动开发（TDD）。TDD中有很多变化，但核心思想是在实现功能本身之前为功能编写测试。首先编写测试可以使你更仔细地考虑正在实现的规范，以及如何知道功能已经正确实现。TDD不是深入实现细节，而是提前考虑成功或失败的度量。

你编写的测试一开始都将失败，因为你的新功能还没有实现。但是随着你为应用程序添加代码，你的测试将逐渐从红色变为绿色，并且在所有功能都完成的时候，测试将全部通过。TDD需要训练，

它确实产生了更全面的测试集，并且生成更可靠的代码。

如果重新运行测试，你将看到它们全部通过，如图7-6所示。

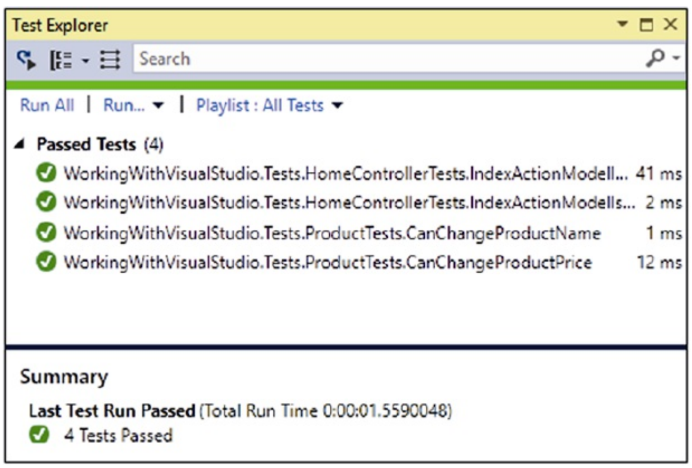


图7-6　所有测试已通过

对于这样一个简单的问题，看起来需要做很多工作，但是在实际的应用程序中，测试特定组件的能力是至关重要的。若已经识别出问题，只有在有效隔离组件的情况下，才能验证和修复程序。

7.3　改进单元测试

7.2节介绍了在Visual Studio中编写和运行单元测试的基本方法，并强调了隔离正在测试的组件的重要性。本节将介绍一些更加高级的工具和特性，你可以使用它们更简洁、更有表现力地编写测试。如果已浸入单元测试文化，那么你最终可以得到很多测试代码，并且代码的清晰性也变得重要起来，特别是当需要修改测试以反映开发过程中应用程序的变化并进入维护阶段时。

7.3.1　参数化单元测试

为HomeController类编写的测试显现出仅针对某些数据值的问题。为了测试这种情况，最终创建了两个类似的测试，每个都有自己的假的存储。这是一种重复的方式，特别是因为这些测试之间唯一的区别是存储中Product对象的Price属性的decimal

值。

xUnit.net提供对参数化测试的支持。其中的测试数据已从测试中删除，以便单个方法可以用于多个测试。在代码清单7-15中，使用参数化测试功能来删除用于HomeController类的测试中的重复数据。

代码清单7-15　在HomeControllerTests.cs文件中参数化单元测试

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        class ModelCompleteFakeRepository : IRepository
 {

            public IEnumerable<Product> Products { get;
 set; }

            public void AddProduct(Product p) {
```

```
                // do nothing - not required for test
            }
        }

        [Theory]
        [InlineData(275, 48.95, 19.50, 24.95)]
        [InlineData(5, 48.95, 19.50, 24.95)]
        public void IndexActionModelIsComplete(decimal
price1, decimal price2,
                decimal price3, decimal price4) {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepository {
                Products = new Product[] {
                    new Product {Name = "P1", Price = p
rice1 },
                    new Product {Name = "P2", Price = p
rice2 },
                    new Product {Name = "P3", Price = p
rice3 },
                    new Product {Name = "P4", Price = p
rice4 },
                }
            };

            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
```

```
                        && p1.Price == p2.Price));
        }
    }
}
```

参数化单元测试使用Theory特性而不是标准的Fact特性，还使用了InLineData特性，以允许为单元测试方法定义参数的指定值。由于C#限制了特性中数据值的表达方式，因此在测试方法上定义了4个decimal参数用于为InlineData特性提供值。可在测试方法中使用decimal值来生成Product对象的数组，用于设置假的存储库对象的Products属性。

每个Inline特性定义了一个单独的单元测试，在Visual Studio测试管理器中显示为单独的项，如图7-7所示。测试管理器中的条目显示了将用于单元测试方法中参数的值。

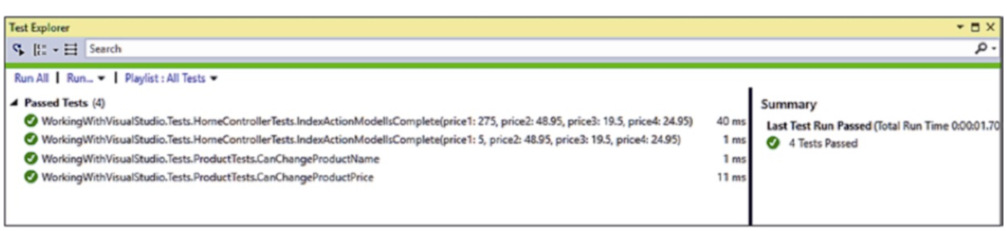


图7-7　Visual Studio测试管理器中的参数化测试

从方法或属性中获取测试数据

在特性中表达数据时会受到限制，这限制了InlineData特性的有效性，但另一种替代方法是创建静态方法或属性来返回测试需要的对象。在这种情况下，对数据的定义方式没有限制，可以创建更广泛的测试值。为演示如何工作，在单元测试项目中添加名为ProductTestData.cs的文件，用它定义代码清单7-16所示的类。

代码清单7-16　WorkingWithVisualStudio.Tests项目中ProductTestData.cs文件的内容

```
using System.Collections;
using System.Collections.Generic;
using WorkingWithVisualStudio.Models;

namespace WorkingWithVisualStudio.Tests {

    public class ProductTestData : IEnumerable<object[]
> {
```

```
        public IEnumerator<object[]> GetEnumerator() {
            yield return new object[] { GetPricesUnder5
0() };
            yield return new object[] { GetPricesOver50
 };
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return this.GetEnumerator();
        }

        private IEnumerable<Product> GetPricesUnder50()
 {
            decimal[] prices = new decimal[] { 275, 49.
95M, 19.50M, 24.95M };
            for (int i = 0; i < prices.Length; i++) {
                yield return new Product { Name = $"P{i
 + 1}", Price = prices[i] };
            }
        }

        private Product[] GetPricesOver50 => new Produc
t[] {
            new Product { Name = "P1", Price = 5 },
            new Product { Name = "P2", Price = 48.95M }
,
            new Product { Name = "P3", Price = 19.50M }
,
            new Product { Name = "P4", Price = 24.95M }
};
    }
}
```

测试数据由实现了IEnumerable<object[]>接口

的类提供，并返回一个对象数组序列。序列中的每个对象数组包含一个将要传递给测试方法的参数集。改进测试方法以便接收Product对象数组，Product对象数组为测试数据添加了另外一层。该层是对象数组的枚举，其中的每个枚举元素都包含一个Product对象数组。测试数据的这种深度结构可能令人困惑，但保证正确是最重要的，因为在Xunit.net试图传递给测试方法的参数个数不匹配方法签名时，测试将不会工作。

构建测试数据类，以便使用私有方法或属性定义单独的测试数据集，然后通过GetEnumerator方法组合到对象数组序列中。为了演示不同的技术，这里使用方法和属性创建了Product对象数组，但是还有另一种方法（选择由正在测试的数据类型来驱动）。代码清单7-17展示了如何通过Theory特性的测试数据类来设置测试。

代码清单7-17　在HomeControllerTests.cs文件中使用测试数据类

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        class ModelCompleteFakeRepository : IRepository
 {

            public IEnumerable<Product> Products { get;
 set; }

            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
        }

        [Theory]
        [ClassData(typeof(ProductTestData))]
        public void IndexActionModelIsComplete(Product[
] products ) {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepository {
                Products = products
            };
```

```
            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }
    }
}
```

提　示

如果希望在与单元测试相同的类中包含测试数据，那么可以使用MemberData特性代替ClassData。MemberData特性使用字符串来配置将提供IEnumerable<object[]>的静态方法名称，序列

中的每个数组可作为测试方法的参数集。

将特性ClassData配置为测试数据类的类型，在本例中是ProductTestData。在运行测试的时候，Xunit.net将创建ProductTestData类的实例，并用来为测试获取测试数据序列。

注　意

在测试管理器中查看测试代码，你将看到IndexActionModelIsComplete测试有单个入口，尽管ProductTestData类提供了两个测试数据集。这发生在测试数据对象不能被序列化的情况下，可以通过

在测试对象上应用Serializable特性来解决。

7.3.2　改进假的实现

隔离组件的有效性需要类的假的实现以提供测试数据，或者检查组件应有的行为。可以创建实现了IRepository接口的类，这是一种有效的方式，但会导致为期望运行的每种测试创建实现类型。作为示例，代码清单7-18展示了为检查在Index操作方法中是否只调用了存储库中Products方法一次所需要做的额外工作（当担心组件对存储库进行重复查询时，这种测试十分常见，会导致多个数据库查询）。

代码清单7-18　为HomeControllerTests.cs文件添加单元测试

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
```

```
using WorkingWithVisualStudio.Models;
using Xunit;
using System;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        class ModelCompleteFakeRepository : IRepository
 {

            public IEnumerable<Product> Products { get;
 set; }

            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
        }

        [Theory]
        [ClassData(typeof(ProductTestData))]
        public void IndexActionModelIsComplete(Product[
] products ) {
            // Arrange
            var controller = new HomeController();
            controller.Repository = new ModelCompleteFa
keRepository {
                Products = products
            };
            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
```

```
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }

        class PropertyOnceFakeRepository : IRepository {

            public int PropertyCounter { get; set; } =
0;

            public IEnumerable<Product> Products {
                get {
                    PropertyCounter++;
                    return new[] { new Product { Name =
 "P1", Price = 100 } };
                }
            }

            public void AddProduct(Product p) {
                // do nothing - not required for test
            }
        }

        [Fact]
        public void RepositoryPropertyCalledOnce() {
            // Arrange
            var repo = new PropertyOnceFakeRepository();

            var controller = new HomeController { Repos
itory = repo };

            // Act
            var result = controller.Index();

            // Assert
            Assert.Equal(1, repo.PropertyCounter);
```

```
        }
    }
}
```

假的实现并不总是简单的数据源，它们也可以用于断言组件执行工作的方式。在这个示例中，添加了一个简单的计数器属性，它在每次读取存储库的Products属性时都会递增，可使用Assert.Equal方法来确保属性仅被调用一次。

1．添加mocking框架

创建假的对象会导致失控，最好的方式是使用假的框架，也就是所谓的mocking框架（假的和仿冒的对象之间有一点技术区别，但是现在，测试工具为易于使用而模糊了它们之间的区别，所以可以交替使用这两个术语）。本章使用的mocking框架名为Moq，其背景如表7-4所示。

表**7-4**　**Moq**框架的背景

问题	答案
是什么？	Moq是一个用于在应用程序中创建组件的仿冒实现的软件包
有何用？	Moq框架使得为单元测试创建仿冒组件以隔离应用程序的部件变得容易
如何使用？	Moq使用Lambda表达式来定义仿冒组件的功能，仅仅需要定义测试可能使用的功能
有何缺陷或限制？	习惯语法可能需要一点努力，请从GitHub网站查看文档和示例
有何替代品？	有多种替代框架可用，包括NSubstitue和FakeItEasy。所有这些框架都提供类似的功能，可在它们之间进行选择

要安装Moq，在Solution Explorer窗格中右击WorkingWithVisualStudio.Tests项目，从弹出的菜单中选择Manage NuGet Packages。单击Browse按钮，在搜索框中输入moq。从包列表中选择Moq，如图7-8所示，单击Install按钮，在项目中添加包。

注　意

把Moq添加到单元测试项目中，而不是应用程序项目中。

2．创建mock对象

创建mock对象意味着告诉Moq期望何种类型的对象，然后配置其行为，将mock对象用于测试的目标。在代码清单7-19中，使用Moq来替换用于测试HomeController的两个假的存储库。

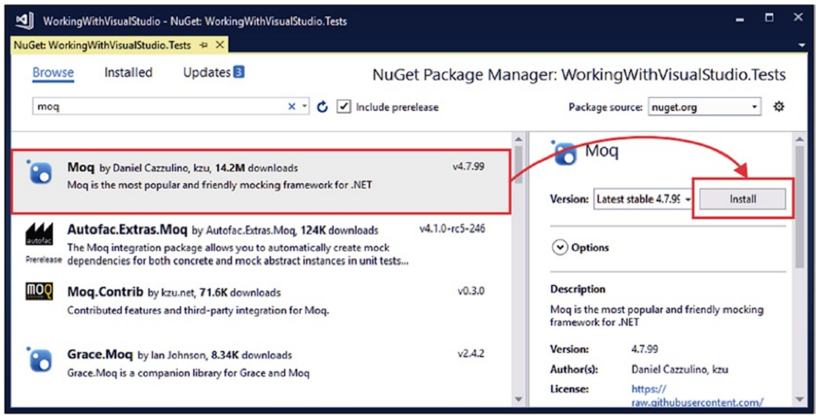


图7-8　为单元测试项目添加包

代码清单7-19　在HomeControllerTests.cs文件中使用mock对象

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;
using System;
using Moq;

namespace WorkingWithVisualStudio.Tests {
    public class HomeControllerTests {

        [Theory]
        [ClassData(typeof(ProductTestData))]
        public void IndexActionModelIsComplete(Product[
] products ) {

            // Arrange
            var mock = new Mock<IRepository>();
            mock.SetupGet(m => m.Products).Returns(prod
ucts);
```

```
            var controller = new HomeController { Repos
itory = mock.Object };

            // Act
            var model = (controller.Index() as ViewResu
lt)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(controller.Repository.Products
, model,
                Comparer.Get<Product>((p1, p2) => p1.Na
me == p2.Name
                    && p1.Price == p2.Price));
        }

        [Fact]
        public void RepositoryPropertyCalledOnce() {

            // Arrange
            var mock = new Mock<IRepository>();
            mock.SetupGet(m => m.Products)
                .Returns(new[] { new Product { Name = "
P1", Price = 100 } });
            var controller = new HomeController { Repos
itory = mock.Object };
            // Act
            var result = controller.Index();

            // Assert
            mock.VerifyGet(m => m.Products, Times.Once);

        }
    }
}
```

使用Moq框架使你可以删除IRepository接口的假的实现，并使用很少几行代码替换它们。本节不会详细介绍Moq框架支持的不同功能，但是会在示例中介绍Moq框架的使用方式（有关Moq框架的示例和文档，参见GitHub网站）。本书的其余内容及示例将说明如何测试不同类型的MVC组件。

第一步是创建一个新的mock对象，指定将要实现的接口，如下所示：

```
...
var mock = new Mock<IRepository>();
...
```

创建的mock对象仿冒了IRepository接口。下一步是定义测试所需的功能。不像常规实现接口的类，mock对象仅仅配置测试所需的行为。对于第一个mock存储库，需要实现Product属性以便返回Product对象的集合并通过ClassData特性传递给测试

方法，如下所示：

```
...
mock.SetupGet(m => m.Products).Returns(products);
...
```

SetupGet方法用于实现属性的读取器。该方法的参数是一个Lambda表达式，用于指定待实现的属性，在示例中是Products。在SetupGet方法的返回结果上调用Returns方法，以指定当属性被读取时返回的结果。对第二个mock存储库使用同样的方法，但是指定固定值，如下所示：

```
...
mock.SetupGet(m => m.Products)
                .Returns(new[] { new Product { Name = "
P1", Price = 100 } });
...
```

Mock类定义了Object属性，用于返回实现了指定接口且带有所定义行为的对象。在这两个单元测试中，使用Object属性获取存储库以配置控制器，

如下所示：

```
...
var controller = new HomeController { Repository = mock
.Object };
...
```

使用的最后一项Moq功能是检查Products属性是否只被调用了一次，如下所示：

```
...
mock.VerifyGet(m => m.Products, Times.Once);
...
```

VerifyGet是一个由Mock类定义的方法，用于在测试完成后检查mock对象的状态。在本例中，VerifyGet方法允许检查Products属性被读取的次数。值Times.Once用于指定如果属性不止读取了一次，就抛出异常，这将导致测试失败（通常，测试中的Assert方法通过在测试失败时抛出异常来工作，这就是VerifyGet方法可以在使用mock对象时替换Assert方法的原因）。

7.4　小结

本章的大部分内容集中于单元测试，单元测试已经成为改进代码质量的强大工具。单元测试不适合每个开发者，但值得尝试。即使仅用于复杂功能或问题诊断也是有用的。本章介绍了xUnit.net框架的使用，说明了测试隔离组件的重要性，演示了简化单元测试代码的一些工具和技术。下一章将介绍如何开发更现实的MVC应用程序，以展示不同功能的组件如何协同工作。

第8章　SportsStore应用程序

前几章构建了快速且简单的MVC应用程序，描述了MVC模式、重要的C#功能以及优秀的MVC开发人员所需要的各种工具。现在是时候组合这些技术以建立简单而实用的电子商务应用程序了。

创建的应用程序名为SportsStore，它遵循一般在线商店的经典做法。首先，创建在线产品目录（客户可以按类别和页面浏览）、购物车（用户可以添加和删除产品以及结账单（供用户输入邮寄详情）。然后，创建一个管理区域，用来管理目录，包括创建、读取、更新和删除（CRUD）功能，这个区域是受保护的，只有登录的管理员可以进行更改。

本章的目的是让你通过创建一个尽可能真实的

例子来了解真正的MVC开发。本书只关注ASP.NET Core MVC，所以简化了与数据库等外部系统的集成，并完全省略了其他部分，如支付处理等。

你可能会发现，建立所需的基础设施部分的过程有些慢，但对MVC应用程序的基础部分进行投入是很有好处的，可以实现可维护、可扩展、结构良好的代码，并对单元测试提供良好的支持。

单元测试

本书已经强调过MVC中单元测试的易用性，以及单元测试如何成为开发过程中非常重要和有用的部分。在本书的这一部分，你将看到这一点，因为这部分已经包含与关键MVC功能相关的单元测试和技术细节。

使用单元测试并不是普遍做法。如果不想进行单元测试，也是可以的。如果你对单元测试不感兴趣，可以跳过这部分，SportsStore应用程序将正常工作。你不需要进行任何类型的单元测试来获得ASP.NET Core MVC的技术优势，当然，支持测试是采用ASP.NET Core MVC的关键原因。

SportsStore应用程序中使用的大多数MVC功能在本书后面也会单独介绍。在这里并不复制所有的部分，这里讲的内容对示例应用程序已经足够了，但会给出对应的其他章节位置，以便你获取更详细的信息。

这里将演示构建应用程序所需的每个步骤，以便你可以看到MVC功能如何组合在一起。当创建视图时，你应该特别注意。如果不严格按照示例，也

许会获得一些奇怪的结果。

8.1　准备开始

如果在阅读本书该部分时，你在自己的计算机上编写SportsStore应用程序，则需要安装Visual Studio，并确保安装LocalDB（这是持久存储数据所必需的）。如果遵循第2章介绍的操作，LocalDB将自动安装在计算机上。

注　意

如果只想在不重新创建项目的情况下执行项目，可以访问Apress网站，上面提供了本书附带的免费源代码，可以从本书的GitHub仓库中下载

SportsStore项目的源代码。当然，你不需要跟着做。作者已尽力使屏幕截图和代码尽可能容易理解。

8.1.1 创建MVC项目

这里将使用与前面章节相同的基本方法，从一个空项目开始，并添加所需的所有配置文件和组件。从Visual Studio的File菜单中选择New→Project，然后选择ASP.NET Core Web Application项目模板，如图8-1所示。将项目的名称设置为SportsStore，然后单击OK按钮。

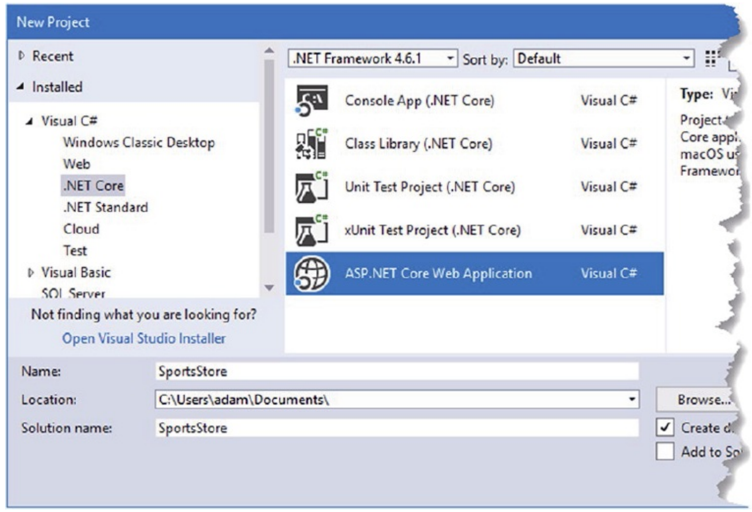


图8-1　选择项目类型

选择Empty模板，如图8-2所示。在单击OK按钮创建SportsStore项目之前，确保在对话框顶部的下拉列表中选择了.NET Core和ASP.NET Core 2.0，并且未勾选Enable Docker Support复选框。

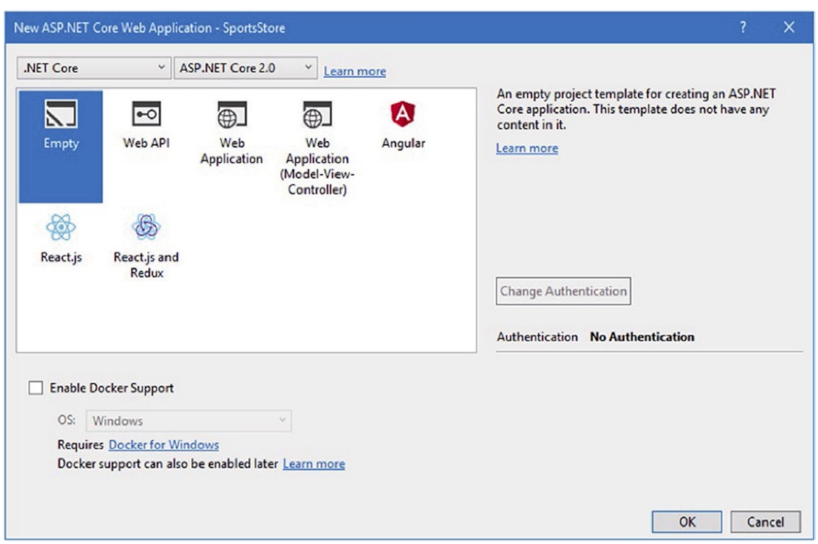


图8-2　选择项目模板

1．创建文件夹结构

下一步是添加一些文件夹，以包含MVC应用程序所需的应用程序组件——模型、控制器和视图。对于表8-1中描述的每个文件夹，右击Solution Explorer窗格中的SportsStore项目，从弹出的菜单中选择Add→New Folder，然后设置文件夹的名称。稍后还需要其他文件夹，但这些文件夹反映了MVC应用程序的主要部分，并且足以开始使用。

表**8-1　SportsStore**项目所需的文件夹

名称	描述
Models	里面包含模型类
Controllers	里面包含控制器类
Views	里面包含与视图相关的所有内容，包括单个Razor文件、视图开始文件和视图导入文件

2．配置应用

Startup类负责配置ASP.NET Core应用程序。代码清单8-1显示了对Startup类所做的更改，以启用MVC框架和一些对开发有用的相关功能。

代码清单8-1　在SportsStore文件夹下的Startup.cs文件中启用功能

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
```

```
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

            });
        }
    }
}
```

注　意

Startup类是一项重要的ASP.NET Core功能，第14章会详细描述它。

ConfigureServices方法用于设置共享对象，可通过依赖注入功能在整个应用程序中使用，第18章

将详细介绍。在ConfigureServices方法中调用的AddMvc方法是一种扩展方法，用于设置MVC应用程序使用的共享对象。

Configure方法用于设置接收和处理HTTP请求的功能。在Configure方法中调用的每种方法都是扩展方法，用于设置HTTP请求处理器，如表8-2所示。

表**8-2**　在**Start**类中调用的初始化方法

方法	描述
UseDeveloperExceptionPage()	显示发生在应用程序中的异常的详细信息，在开发过程中很有帮助。在发布的应用程序中不应该启用，在第12章中发布应用程序时会禁用该特性
UseStatusCodePages()	在HTTP响应中添加一条简单的信息，否则不会有响应体部分，例如，404 - Not Fund响应
UseStaticFiles()	启用对wwwroot文件夹中静态内容的支持
UseMvc()	启用ASP.NET Core MVC

接下来，需要为应用程序准备Razer视图。右击Views文件夹，从弹出菜单中选择Add→New Item，从ASP.NET分类中选择MVC View Imports Page项目，如图8-3所示。

单击Add按钮以创建_ViewImports.cshtml文件，如代码清单8-2所示，设置新文件的内容。

代码清单8-2　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using SportsStore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

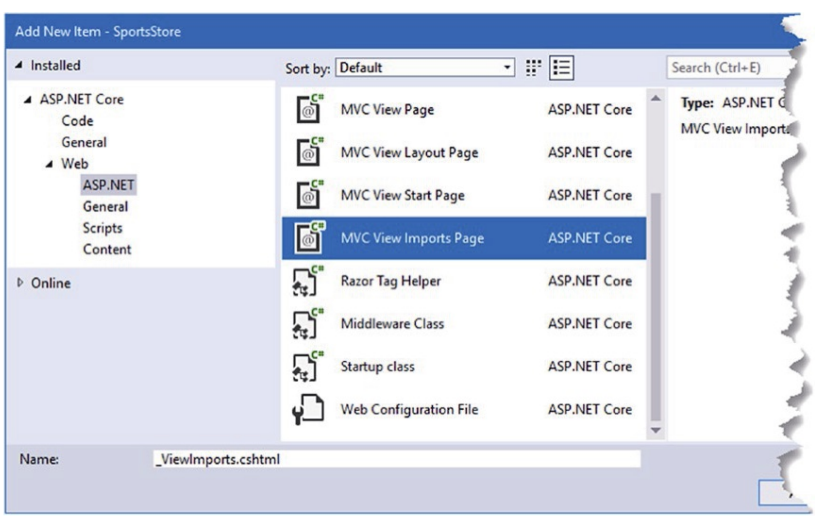


图8-3　创建视图导入文件

语句@using允许在视图中使用定义在命名空间SportsStore.Models中的类型，而不再需要引用命名空间。而语句@addTagHelper启用了内置的标签助手特性，随后使用它来创建反映SportsStore应用程序的配置的HTML元素。

8.1.2　创建单元测试项目

创建单元测试项目的过程与第7章描述的相同。右击Solution Explorer窗格中的SportsStore项，然后从弹出的菜单中选择Add→New Project。选择xUnit Test Project（NET Core）项目模板，如图8-4所示，并将项目名称设置为SportsStore.Tests。单击OK按钮即可创建单元测试项目。

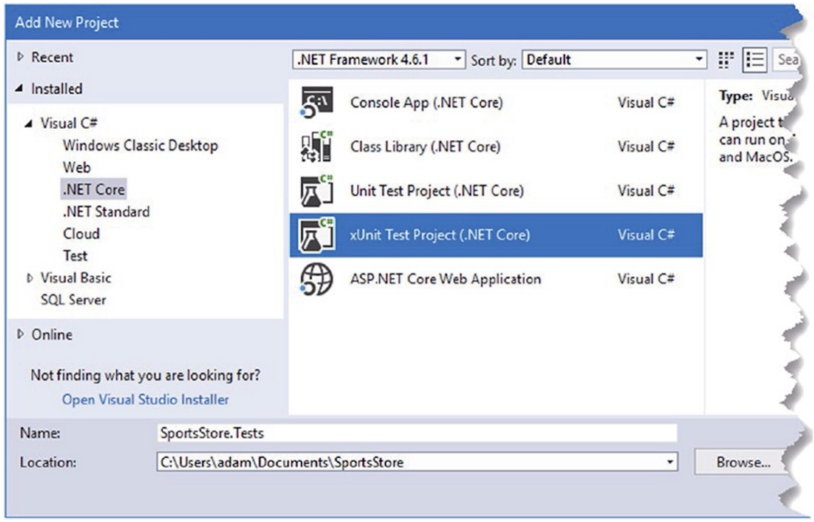


图8-4　选择项目模板

一旦创建了单元测试项目，就右击解决方案资源管理器中的SportsStore.Tests项目并从弹出菜单中选择Edit SportsStore.Tests.csproj。添加代码清单8-3中加粗显示的新元素，以将Moq包添加到测试项目中，并创建对SportsStore项目的引用。确保为Moq包指定了列表中显示的版本。

代码清单8-3　在SportsStore.Tests项目的SportsStore.Tests.csproj文件中添加包

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\SportsStore\SportsSto
re.csproj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
        Version="15.3.0-preview-20170628-02" />
    <PackageReference Include="xunit" Version="2.2.0" /
>
    <PackageReference Include="xunit.runner.visualstudi
o" Version="2.2.0" />
    <PackageReference Include="Moq" Version="4.7.99" />
  </ItemGroup>

</Project>
```

在将更改保存到.csproj文件时，Visual Studio将下载Moq包并将其安装到单元测试项目中，并创建对SportsStore项目的引用，以便其中包含的类可用于测试。

8.1.3 测试和启动应用程序

现在已经创建并配置了应用程序和单元测试项目，可以进行开发了。Solution Explorer窗格应包含图8-5所示的项目。如果看到不同的项目或项目不在同一位置，你将会遇到问题，所以请花一点时间检查这些项目是否存在，并已位于正确的位置。

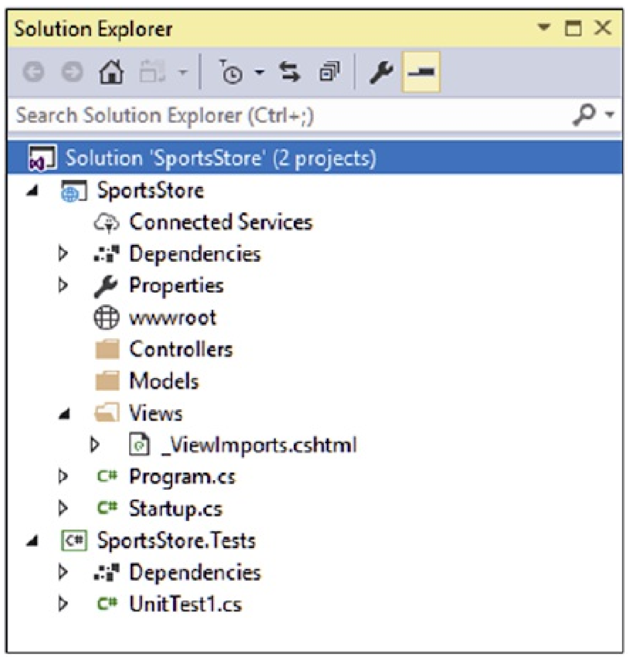


图8-5　用于SportsStore应用程序和单元测试项目的Solution Explorer窗格

如果从Debug菜单中选择Start Debugging（或者选择Start Without Debugging），你将看到错误页

面，如图8-6所示。显示错误消息是因为应用程序中没有控制器来处理当前的请求，作者将很快解决这个问题。

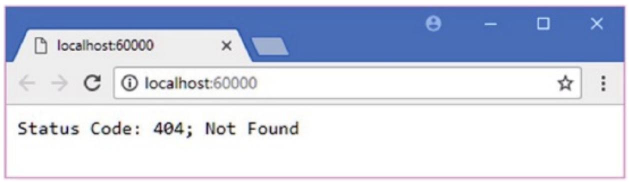


图8-6　运行SportsStore应用程序后的错误页面

8.2　开始领域模型开发

所有项目都是从MVC应用程序的核心——领域模型开始的。这是一个电子商务应用程序，因此最需要的模型就是产品。添加名为Product.cs的类文件到Models文件夹中，并用它定义代码清单8-4所示的类。

代码清单8-4　Models文件夹下的Product.cs文件的内容

```
namespace SportsStore.Models {
```

```
    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

8.2.1　创建存储库

现在需要一些从数据库获取Product对象的方法。正如第3章所解释的，模型包括用于从持久化数据库中存储和检索数据的逻辑。此时本书并不关心如何实现数据持久性，但本书将为之定义接口。在Models文件夹中添加名为IProductRepository.cs的C#接口文件，并用它定义代码清单8-5所示的接口。

代码清单8-5　Models文件夹下的IProductRepository.cs文件的内容

```
using System.Linq;
```

```
namespace SportsStore.Models {

    public interface IProductRepository {

        IQueryable<Product> Products { get; }
    }
}
```

这个接口使用IQueryable<T>来允许调用者获得一系列Product对象。IQueryable<T>接口来自你更熟悉的IEnumerable<T>接口，表示可以查询的对象集合，例如由数据库管理的对象。

依赖于IProductRepository接口的类可以获取Product对象，而无须知道它们的存储方式或实现类将会如何传递它们的详细信息。

IEnumerable <T>和**IQueryable <T>**接口

IQueryable<T>接口很有用，因为它允许有效地

查询对象集合。在本章的后面，将添加对从数据库中检索产品对象子集的支持，并且IQueryable<T>接口允许使用标准LINQ语句向数据库查询需要的对象，而无须知道数据库服务器存储数据或处理查询的方式。如果没有IQueryable<T>接口，本书将不得不从数据库中检索所有Product对象，然后丢弃那些不想要的对象，随着应用程序使用的数据量的增加，代价将变得非常昂贵。正因为如此，数据库存储库接口和类通常使用IQueryable<T>接口而不是IEnumerable<T>接口。但是，必须小心使用IQueryable<T>接口，因为每次枚举对象集合时，都会再次评估查询，这意味着将向数据库发送新的查询。这可能会破坏使用IQueryable<T>的好处。在这种情况下，可以使用ToList或ToArray扩展方法将IQueryable<T>转换为更可预测的形式。

8.2.2　创建虚拟存储库

既然已经定义了一个接口，就可以实现持久化机制并连接到一个数据库，但是这里希望先添加应用程序的其他部分。为此，创建IProductRepository接口的虚拟实现，等到再次处理数据存储的时候再真正实现。为了创建虚拟存储库，在Models文件夹中添加名为FakeProductRepository.cs的文件，并用它定义代码清单8-6所示的类。

代码清单8-6　Models文件夹下的FakeProductRepository.cs文件的内容

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class FakeProductRepository : IProductReposi
tory {

        public IQueryable<Product> Products => new List
<Product> {
            new Product { Name = "Football", Price = 25
 },
```

```
            new Product { Name = "Surf board", Price = 
179 },
            new Product { Name = "Running shoes", Price
 = 95 }
        }.AsQueryable<Product>();
    }
}
```

FakeProductRepository类返回固定的Product对象集合，并作为Products属性的值，以此实现IProductRepository接口。实现IProductRepository接口时还需要AsQueryable方法，用于将固定的对象集合转换为IQueryable <Product>，并允许创建兼容的虚拟存储库而不必处理真正的查询。

8.2.3　注册存储库服务

MVC 强调使用松耦合组件，这意味着在更改应用程序的某个部分后，无须在其他位置进行相应的更改。这会将应用程序的一部分变成服务，这些服务提供了应用程序其他部分需要使用的功能。这些类提供的服务可以更改或替换，而无须更改使用

它们的类。第18章将深入解释这一点，但对于SportsStore应用程序，作者希望创建存储库服务，以允许控制器获取实现了IProductRepository接口的对象，而不需要知道正在使用哪个类。可以使用之前创建的FakeProductRepository类开发应用程序，然后替换为真正的存储库，而无须对需要访问存储库的所有类进行更改。服务已在Startup类的ConfigureServices方法中注册，在代码清单8-7中，为存储库定义了新的服务。

代码清单8-7　在SportsStore文件夹下的Startup.cs文件中创建存储库服务

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;

namespace SportsStore {
```

```
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IProductRepository, F
akeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

            });
        }
    }
}
```

添加到ConfigureServices方法的语句告诉ASP.NET Core当组件（如控制器）需要IProductRepository接口的实现时，应该会收到一个FakeProductRepository对象。AddTransient方法指定了一个新的FakeProductRepository对象，每次需要IProductRepository接口时都应创建它。现在这段代

码还没有任何意义，不要担心；你很快就会看到如何将之应用于应用程序，第18章将详细介绍这些情况。

8.3　显示产品清单

可以在本章的其余部分构建领域模型和存储库，而不触及应用程序的其余部分。不过，你可能会觉得无聊，所以想要切换一下，开始认真使用MVC，并根据需要添加模型和存储库。

在本节中，将创建一个控制器和一个操作方法，可以在存储库中显示产品的详细信息。目前，这只是为了显示虚拟存储库中的数据，但稍后会对其进行排序。这里还将设置初始路由配置，以便MVC知道如何将应用程序的请求映射到创建的控制器。

使用**Visual Studio MVC**脚手架

在本书中，都是通过右击Solution Explorer窗格中的文件夹来创建MVC控制器和视图，方法是从弹出菜单中选择Add→New Item，然后在Add New Item窗口中选择模板。还有一种替代方案，称为scaffolding（脚手架），Visual Studio在Add菜单中提供了专门用于创建控制器和视图的菜单项。当选择这些菜单项时，需要为创建的组件选择场景，例如具有读/写操作的控制器或包含用于创建特定模型对象的表单视图。

本书不使用脚手架。脚手架生成的代码和标记是非常通用的，但并不是很有用，因为这些代码和标记只能支持一组特定的场景，不能解决常见的开发问题。本书的目标不仅在于确保你知道如何创建

MVC应用程序，还需要解释幕后的一切工作。如果将创建组件的责任交给脚手架，解释起来将会更加困难。

也就是说，你的开发风格可能与作者的不同，并且你可能更喜欢在自己的项目中使用脚手架，这将是另一种情况。这是非常合理的，但是建议你花时间了解脚手架的作用，以便在没有得到预期结果的情况下知道去哪里检查。

8.3.1　添加一个控制器

为了在应用程序中创建第一个控制器，将名为ProductController.cs的类文件添加到Controllers文件夹中，并定义代码清单8-8所示的类。

代码清单8-8　Controller文件夹下的ProductController.cs文件的

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }
    }
}
```

当MVC需要创建一个新的ProductController实例来处理HTTP请求时，它将检查构造函数，并看到需要一个实现了IProductRepository接口的对象。要确定应该使用什么实现类，MVC会参考Startup类中的配置，得知应该使用FakeRepository，并且每次都应该创建一个新的实例。MVC会创建一个新的FakeRepository对象，并使用它调用ProductController构造函数，以创建用于处理HTTP

请求的控制器对象。

这称为依赖注入（dependency injection），这种方式允许ProductController通过IProductRepository接口访问应用程序的存储库，而无须了解配置了哪个实现类。之后，将使用真实存储库替换虚拟存储库，使用依赖注入意味着控制器将继续工作而无须做任何更改。

注　意

一些开发者不喜欢依赖注入，并认为它会使应用程序更加复杂。作者不这样认为，但如果你刚开始使用依赖注入，那么建议你看完第18章后，再来决定是否使用。

接下来，添加一个名为List的操作方法，它将呈现一个视图，以显示存储库中产品的完整列表，如代码清单8-9所示。

代码清单8-9　在Controller文件夹下的ProductController.cs文件中添加一个操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }

        public ViewResult List() => View(repository.Pro
ducts);
    }
}
```

像这样调用View方法（不指定视图名称）会告诉MVC渲染操作方法的默认视图。将List<Product>（Product对象列表）传递给View方法，从而为框架提供在强类型视图中填充Model对象的数据。

8.3.2　添加并配置视图

需要创建视图以呈现用户的内容，还需要执行一些准备步骤来让视图更简单。首先，创建共享布局，定义将被包含在发送给客户端的所有HTML响应中的通用内容。共享布局是确保视图一致并包含重要的JavaScript文件和CSS样式表的有效方法，第5章解释了它们的工作原理。

创建Views/Shared文件夹，并添加名为_Layout.cshtml的MVC视图布局页面，_Layout.cshtml是Visual Studio分配给这类项目的默

认名称。代码清单8-10显示了_Layout.cshtml文件的内容，其中对默认内容进行了更改，将title元素的内容设置为SportsStore。

代码清单8-10　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>SportsStore</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

接下来，需要配置应用程序，以便默认应用_Layout.cshtml文件，这可通过将名为_ViewStart.cshtml的MVC视图启动文件添加到

Views文件夹来完成。Visual Studio添加的默认内容如代码清单8-11所示，选择名为_Layout.cshtml的布局，该布局对应代码清单8-11所示的文件。

代码清单8-11　Views文件夹下的_ViewStart.cshtml文件的内容

```
@{
    Layout = "_Layout";
}
```

现在需要添加一个视图，当名为List的操作方法用于处理请求时将显示这个视图。创建Views/Product文件夹，并添加名为List.cshtml的Razor视图文件，然后添加代码清单8-12所示的标记。

代码清单8-12　Views/Product文件夹下的List.cshtml文件的内容

```
@model IEnumerable<Product>

@foreach (var p in Model) {
    <div>
```

```
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

文件顶部的@model表达式指定视图将从操作方法中接收一系列Product对象作为模型数据。使用@foreach表达式来处理序列，并为接收的每个Product对象生成一组简单的HTML元素。

视图不知道Product对象来自哪里，如何获取，以及它们是否代表应用程序已知的所有产品。相反，视图仅涉及使用HTML元素显示每个产品的详细信息，这与第3章描述的关注点分离是一致的。

提　示

可使用.ToString（"c"）方法将Price属性转换为字符串，该方法会根据服务器上有效的culture设置将数值作为货币进行转换。例如，如果服务器设置为en-US，（1002.3）.ToString（"c"）将返回$1,002.30；但如果服务器设置为en-GB，那么相同的方法将返回£1,002.30。

8.3.3　设置默认路由

需要告诉MVC它应该将访问应用程序的根URL（****://mysite/）的请求发送到ProductController类的名为List的操作方法。可通过编辑Startup类中的语句来设置处理HTTP请求的MVC类，如代码清单8-13所示。

代码清单8-13　更改SportStore文件夹下的Startup.cs文件中的默认路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;

namespace SportsStore {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IProductRepository, F
akeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{ac
tion=List}/{id?}");
            });
        }
    }
}
```

Startup类的Configure方法用于设置请求管道，由检查HTTP请求并生成响应的类（称为中间件）组成。UseMvc方法用于设置MVC中间件，其中一个配置选项用于将URL映射到控制器和操作方法。第15章和第16章将详细描述路由系统，但代码清单8-13告诉MVC将请求发送到Product控制器的名为List的操作方法，除非请求URL另有指定。

提　示

需要将代码清单8-13中的控制器名称设置为Product而不是控制器类的名称ProductController，这是MVC命名约定的一部分，其中控制器类的名称通常以Controller结尾，但是在引用类时会忽略这部

分名称。第31章将解释命名约定及其作用。

8.3.4　运行应用程序

现在所有基础工作已准备就绪。创建了一个控制器，其中包含一个操作方法，当请求应用程序的默认URL时，MVC将使用这个操作方法。MVC将创建一个FakeRepository实例，并使用它创建一个新的控制器对象来处理请求。虚拟存储库将为控制器提供一些简单的测试数据，将操作方法传递给Razor视图，以便在发送给浏览器的HTML响应中包含每个产品的详细信息。当生成HTML响应时，MVC将来自操作方法所选视图的数据与来自共享布局的内容相结合，生成浏览器可以解析和显示的完整HTML文档。可以通过启动应用程序来查看结果，如图8-7所示。

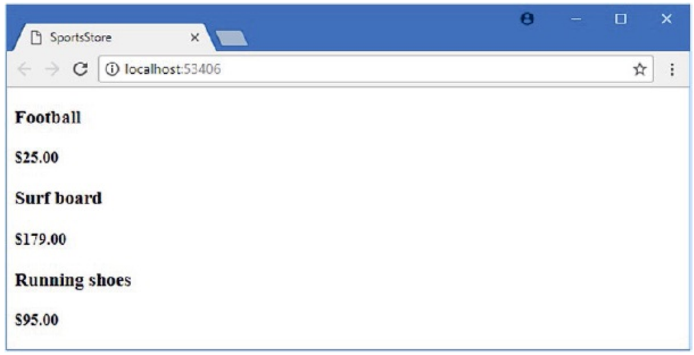


图8-7　查看基本的应用程序功能

这是ASP.NET Core MVC的典型开发模式。虽然需要花费一点时间才能完成所有的初始设置，但这是必要的，这样应用程序的基本功能才能快速集成在一起。

8.4　准备数据库

可以显示包含产品详细信息的简单视图，但使用虚拟存储库提供的测试数据。在使用真实数据实现真正的存储库之前，需要建立数据库并用一些数据填充。

这里将使用SQL Server作为数据库，并且使用Entity Framework Core（EF Core）（Microsoft.NET对象关系映射（ORM）框架）访问数据库。ORM框架通过常规C#对象呈现关系数据库的表、列和行。

注　意

可以从各种工具和技术中进行选择。不仅可以使用不同的关系数据库，还可以使用对象存储库、文档存储和其他一些更复杂的备选方案。甚至可以使用其他的.NET ORM框架，每个框架都采用稍微不同的实现方式；这些变化可能更适合你的项目。

这里选择Entity Framework Core有几个原因：使用起来简单，与LINQ的集成非常好，并且能够与ASP.NET Core MVC很好地协作。早期的版本多少有点问题，但是现在的版本非常优雅，而且功能丰富。

SQL Server有一个很好的功能——LocalDB，它是专为开发人员设计的，具有基本的SQL Server功能，并且不需要管理功能。有了LocalDB，就可以在构建项目时跳过设置数据库的步骤，稍后再部署到完整的SQL Server实例。大多数MVC应用程序被部署到由专业管理人员管理的托管环境中，因此LocalDB功能意味着数据库配置可以留给DBA来完成，开发人员可以继续进行编码。

提　示

如果在安装Visual Studio时没有选择LocalDB，那么现在需要安装它，可以通过Visual Studio安装程序的Individual Components部分进行选择。如果按照第2章中的说明执行操作，那么LocalDB功能应该已经安装好并可以使用了。

8.4.1　安装Entity Framework Core工具包

当使用Visual Studio创建项目时，默认已将主要的Entity Framework Core功能添加到项目中。你还需要一个额外的NuGet包来提供命令行工具，这些工具用于创建一些类，从而准备数据库并存储应用程序的数据（称为迁移）。

要将包添加到项目中，请右击Solution Explorer

窗格中的SportsStore项，从弹出菜单中选择Edit SportsStore.csproj，然后根据代码清单8-14所示的文件进行更改。请注意，使用列表中指定的版本，并使用DotNetCliToolReference元素添加包，而不是使用PackageReference元素。

代码清单8-14　在SportsStore文件夹下的SportsStore.csproj文件中添加包

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFr
ameworkCore.Tools.DotNet"
        Version="2.0.0" />
  </ItemGroup>

</Project>
```

注　意

必须通过编辑文件来安装这个包。这种类型的包无法使用NuGet Package Manager或dotnet命令行工具进行添加。

当保存文件时，Visual Studio将下载并安装Entity Framework Core命令行工具，然后添加到项目中。

8.4.2　创建数据库类

在Models文件夹中添加名为

ApplicationDbContext.cs的类文件，并定义代码清单8-15所示的类。数据库上下文类（database context class）是应用程序和Entity Framework Core之间的桥梁，它使用模型对象提供对应用程序数据的访问。为了创建SportsStore应用程序的数据库上下文类，在Models文件夹中添加名为ApplicationDbContext.cs的类文件，并定义代码清单8-15所示的类。

代码清单8-15　Models文件夹下的ApplicationDbContext.cs文件的内容

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions<Ap
plicationDbContext> options)
            : base(options) { }

        public DbSet<Product> Products { get; set; }
```

```
    }
}
```

DbContext基类提供对Entity Framework Core的底层功能的访问，Products属性将提供对数据库中Product对象的访问。ApplicationDbContext类从DbContext派生，并添加了一些属性，用于读取和写入应用程序数据。目前只有一个属性，它将提供对Product对象的访问。

8.4.3　创建存储库类

目前建立数据库所需的大部分工作已完成。下一步是创建一个实现了 IProductRepository 接口并使用Entity Framework Core获取数据的类。在Models文件夹中添加名为EFProductRepository.cs的类文件，并定义代码清单8-16所示的存储库类。

代码清单8-16　Models文件夹下的EFProductRepository.cs文件的内容

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class EFProductRepository : IProductReposito
ry {
        private ApplicationDbContext context;
        public EFProductRepository(ApplicationDbContext
 ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.
Products;
    }
}
```

这里将向应用程序添加其他功能，但目前，存储库实现只是将由IProductRepository接口定义的Products属性映射到由ApplicationDbContext类定义的Products属性。上下文类中的Products属性会返回一个DbSet <Product>对象，该对象实现了IQueryable<T>接口，并且在使用Entity Framework Core时可以轻松实现IProductRepository接口。这可以确保对数据库的查询将只检索所需的对象，如本

章前面所述。

8.4.4　定义连接字符串

连接字符串（connection string）用来指定数据库的位置和名称，并提供应用程序如何连接到数据库服务器的配置设置。连接字符串存储在名为appsettings.json的JSON文件中，在SportsStore项目中，可使用Add New Item窗口的General部分的ASP.NET Configuration File模板创建该文件。

Visual Studio在创建文件时会向appsettings.json文件添加占位符连接字符串，可在代码清单8-17中进行替换。

代码清单8-17　编辑SportsStore文件夹下的appsettings.json文件中的连接字符串

```
{
  "Data": {
    "SportStoreProducts": {
```

```
      "ConnectionString": "Server=(localdb)\\MSSQLLocal
DB;Database=SportsStore;Trusted_Conne
      ction=True;MultipleActiveResultSets=true"
    }
  }
}
```

提　示

连接字符串必须表示为完整的行，这在Visual Studio编辑器中是没问题的，但不适合图书版面，因而代码清单8-17中的格式比较奇怪。当在自己的项目中定义连接字符串时，请确保ConnectionString的值位于同一行。

在配置文件的Data部分，已将连接字符串的名

称设置为SportsStoreProducts。ConnectionString的值表示应将LocalDB功能用于名为SportsStore的数据库。

8.4.5　配置应用程序

接下来的步骤是读取连接字符串，并配置应用程序以连接到数据库。代码清单8-18显示了对Startup类所做的更改，以接收appsettings.json文件中包含的配置数据的详细信息，并用来配置Entity Framework Core（读取JSON文件的工作由Program类处理，这将在第14章中介绍）。

代码清单8-18　在SportsStore文件夹下的Startup.cs文件中配置应用程序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));
            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{ac
```

```
tion=List}/{id?}");
            });
        }
    }
}
```

添加到Startup类的构造函数会接收从appsettings.json文件加载的配置数据，该文件可通过实现了IConfiguration接口的对象来呈现。Startup构造函数将IConfiguration对象分配给名为Configuration的属性，以便Startup类的其余部分可以使用。

第14章将解释如何读取和访问配置数据。对于SportsStore应用程序，添加一系列方法调用，用于在ConfigureServices方法中设置Entity Framework Core。

```
...
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration["Data:SportStore
Products:ConnectionString"]));
...
```

AddDbContext扩展方法用于为代码清单8-15中创建的数据库上下文类设置由Entity Framework Core提供的服务。正如在第14章中解释的那样，在Startup类中使用的许多方法允许使用选项参数来配置服务和中间件功能。AddDbContext方法的参数是一个Lambda表达式，它接收为上下文类配置数据库的可选对象。在这种情况下，使用UseSqlServer方法配置数据库，并指定从Configuration属性获取的连接字符串。

在Startup类中做出的下一个更改是用真实存储库替换虚拟存储库，如下所示：

```
...
services.AddTransient<IProductRepository, EFProductRepository>();
...
```

使用了IProductRepository接口的应用程序中的组件（目前只是Product控制器）将在创建时接收

EFProductRepository对象，从而使它们可以访问数据库中的数据。第18章将详细解释这是如何工作的，结果是模拟数据将被数据库中的真实数据无缝替换，而无须更改ProductController类。

禁用范围验证

使用Entity Framework Core需要对依赖注入功能的配置进行更改，具体方法将在第18章中介绍。在将控制权交给Startup类之前，Program类负责启动和配置ASP.NET Core，代码清单8-19显示了需要执行的更改。如果不做更改，那么在尝试创建数据库架构时将引发异常。

代码清单8-19　在SportsStore文件夹下的Program.cs文件中准备使用Entity Framework Core

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
namespace SportsStore {
    public class Program {
        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .UseDefaultServiceProvider(options =>
                    options.ValidateScopes = false)
                .Build();
    }
}
```

第14章将解释如何详细配置ASP.NET Core，但这是SportsStore应用程序所需的对Program类要做的唯一更改。

8.4.6　创建数据库迁移

Entity Framework Core能够通过称为迁移

（migration）的功能，使用模型类生成数据库架构。准备迁移时，Entity Framework Core会创建一个包含准备数据库所需的SQL命令的C#类。如果需要修改模型类，则可以创建一个新的迁移，其中包含反映更改所需的SQL命令。通过这种方式，不必担心手动编写和测试SQL命令，而只需关注应用程序中的C#模型类。

Entity Framework Core命令从命令行执行。打开一个新的命令提示符或PowerShell窗口，导航到SportsStore项目文件夹（包含Startup.cs和appsettings.json文件的文件夹），然后运行以下命令来创建迁移类，以准备第一次使用数据库：

```
dotnet ef migrations add Initial
```

执行完以上命令后，你将在Visual Studio的Solution Explorer窗格中看到Migrations文件夹，这

是Entity Framework Core存储其迁移类的地方。其中一个文件名将是一个时间戳，后跟_Initial.cs，这是用于为数据库创建初始架构的类。如果查看此文件的内容，可以看到如何使用Product模型类来创建数据库架构。

Add-Migration和Update-Database命令

对于经验丰富的Entity Framework开发者，则可能习惯使用Add-Migration命令来创建数据库迁移，并使用Update-Database命令将其应用于数据库。

随着.NET Core的推出，Entity Framework Core添加了集成到dotnet命令行工具的命令，可通过添加到代码清单8-14中的Microsoft.EntityFrameworkCore.Tools.DotNet软件包

来使用。本章将使用这些命令，因为它们与其他.NET命令一致，可以在任何命令提示符或PowerShell窗口中使用，而不像Add-Migration和Update-Database命令，它们仅在特定Visual Studio窗口中使用。

8.4.7　创建种子数据

为了填充数据库并提供一些示例数据，在Models文件夹中添加名为SeedData.cs的类文件，并定义代码清单8-20所示的类。

代码清单8-20　Models文件夹下的SeedData.cs文件的内容

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
```

```
    public static class SeedData {

        public static void EnsurePopulated(IApplication
Builder app) {
            ApplicationDbContext context = app.Applicat
ionServices
                .GetRequiredService<ApplicationDbContex
t>();
            context.Database.Migrate();
            if (!context.Products.Any()) {
                context.Products.AddRange(
                    new Product {
                        Name = "Kayak", Description = "
A boat for one person",
                        Category = "Watersports", Price
 = 275 },
                    new Product {
                        Name = "Lifejacket",
                        Description = "Protective and f
ashionable",
                        Category = "Watersports", Price
 = 48.95m },
                    new Product {
                        Name = "Soccer Ball",
                        Description = "FIFA-approved si
ze and weight",
                        Category = "Soccer", Price = 19
.50m },
                    new Product {
                        Name = "Corner Flags",
                        Description = "Give your playin
g field a professional touch",
                        Category = "Soccer", Price = 34
.95m },
                    new Product {
                        Name = "Stadium",
```

```
                        Description = "Flat-packed 35,0
00-seat stadium",
                        Category = "Soccer", Price = 79
500 },
                    new Product {
                        Name = "Thinking Cap",
                        Description = "Improve brain ef
ficiency by 75%",
                        Category = "Chess", Price = 16
},
                    new Product {
                        Name = "Unsteady Chair",
                        Description = "Secretly give yo
ur opponent a disadvantage",
                        Category = "Chess", Price = 29.
95m },
                    new Product {
                        Name = "Human Chess Board",
                        Description = "A fun game for t
he family",
                        Category = "Chess", Price = 75
},
                    new Product {
                        Name = "Bling-Bling King",
                        Description = "Gold-plated, dia
mond-studded King",
                        Category = "Chess", Price = 120
0
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

静态的EnsurePopulated方法会接收IApplicationBuilder参数，该参数是Startup类的Configure方法中使用的接口，用来注册中间件以处理HTTP请求，这是确保数据库具有内容的地方。

EnsurePopulated方法通过IApplicationBuilder接口获取ApplicationDbContext对象，并调用Database.Migrate方法以确保已应用迁移，这意味着将创建和准备数据库，以便可以存储Product对象。接下来，检查数据库中Product对象的数量。如果数据库中没有对象，则使用AddRange方法，用一组Product对象填充数据库，然后使用SaveChanges方法将它们写入数据库。

最后的更改是在应用程序启动时对数据库填充种子数据，可通过从Startup类中调用EnsurePopulated方法来完成，如代码清单8-21所示。

代码清单8-21　在SportsStore文件夹下的Startup.cs文件中对数据库填充种子数据

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));
            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{ac
tion=List}/{id?}");
            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

启动应用程序，创建数据库并为其填充种子数据，用于向应用程序提供数据（请耐心等待，可能需要一些时间才能创建数据库）。

当浏览器请求应用程序的默认URL时，应用程序配置告知MVC需要创建一个Product控制器来处理请求。创建新的Product控制器意味着调用ProductController构造函数，该构造函数需要一个实现了IProductRepository接口的对象，新的配置告诉

MVC应该为此创建并使用EFProductRepository对象。EFProductRepository对象利用Entity Framework Core功能，从SQL Server加载数据并将其转换为Product对象。所有这些对ProductController类都是隐藏的，该类只接收一个实现了IProductRepository接口的对象，并使用其提供的数据。结果是在浏览器中显示数据库中的样本数据，如图8-8所示。

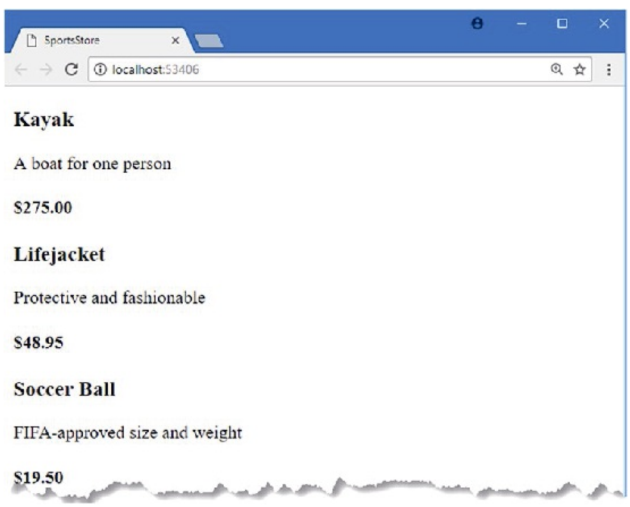


图8-8　数据库中的样本数据

这种让Entity Framework Core将SQL Server数据库呈现为一系列模型对象的方法非常简单且易于使

用，使你能够将注意力集中在ASP.NET Core MVC上，从而跳过Entity Framework Core运行的大量细节以及可用的大量配置选项。作者非常喜欢Entity Framework Core，建议你花一些时间详细了解它。一个很好的起点是微软的Entity Framework Core网站。

8.5　添加分页

从图8-8可以看出，List.cshtml视图文件在单个页面上显示了数据库中的产品。在本节中，将添加对分页的支持，以便视图在页面上显示较少数量的产品，用户可以翻页查看全部目录。为此，在Product 控制器的List方法中添加一个参数，如代码清单8-22所示。

代码清单8-22　在Controller文件夹下，为ProductController.cs文件中名为List的操作方法添加分页支持

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }

        public ViewResult List(int productPage = 1)
            => View(repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((productPage - 1) * PageSize)
                .Take(PageSize));
    }
}
```

PageSize字段指定每页需要4个产品。这里向List方法添加了一个可选参数，从而意味着如果调用没有参数的方法（List()），调用将被视为已经提供了参数定义中指定的值（List（1））。结果是，当MVC调用不带参数的List方法时，将显示产品的第一页。在操作方法的主体中，得到了Product对

象，可通过主键对它们进行排序，跳过当前页面之前出现的产品，并获取PageSize字段指定的产品数量。

单元测试——分页

可以通过创建模拟的存储库，再注入ProductController类的构造函数，并调用List方法来请求特定的页面来对分页功能进行单元测试。然后，可以对得到的Product对象与期望从模拟实现的测试数据中得到的结果进行比较。有关如何设置单元测试的详细信息，请参阅第7章。这里是为此创建的单元测试，已添加到SportsStore项目的名为ProductControllerTests.cs的类文件中。

```
using System.Collections.Generic;
using System.Linq;
using Moq;
```

```
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class ProductControllerTests {

        [Fact]
        public void Can_Paginate() {
            // Arrange
            Mock<IProductRepository> mock = new Mock<IP
roductRepository>();
            mock.Setup(m => m.Products).Returns((new Pr
oduct[] {
                new Product {ProductID = 1, Name = "P1"
},
                new Product {ProductID = 2, Name = "P2"
},
                new Product {ProductID = 3, Name = "P3"
},
                new Product {ProductID = 4, Name = "P4"
},
                new Product {ProductID = 5, Name = "P5"
}
            }).AsQueryable<Product>());

            ProductController controller = new ProductC
ontroller(mock.Object);
            controller.PageSize = 3;

            // Act
            IEnumerable<Product> result =
                controller.List(2).ViewData.Model as IE
numerable<Product>;
```

```
            // Assert
            Product[] prodArray = result.ToArray();
            Assert.True(prodArray.Length == 2);
            Assert.Equal("P4", prodArray[0].Name);
            Assert.Equal("P5", prodArray[1].Name);
        }
    }
}
```

从操作方法返回的数据不太好处理。结果是一个ViewResult对象，必须将ViewData.Model属性的值转换为预期的数据类型。第17章将解释由操作方法返回的不同结果类型以及如何使用它们。

8.5.1 显示页面链接

如果运行应用程序，你将看到页面上显示了4个条目。如果要查看其他页面，可以将查询字符串参数附加到URL的末尾，如下所示：

```
http://localhost:5000/?productPage=2
```

你需要更改URL的端口部分，以匹配项目使用的端口。使用这些查询字符串，可以浏览产品目录。

客户无法确定这些查询字符串参数是否存在，即使它们存在，也不会以这种方式导航。相反，需要在每个产品列表的底部呈现一些页面链接，以便客户可以在页面之间导航。为了做到这一点，需要实现一个标签助手（tag helper），它可以为所需链接生成HTML标记。

1．添加视图模型

为了使用标签助手，需要向视图传递可用页面的数量、当前页面以及存储库中产品总数的信息。最简单的方法是创建视图模型类，专门用于在控制器和视图之间传递数据。在SportsStore项目中创建Models/ViewModels文件夹，并添加名为

PagingInfo.cs的类文件，如代码清单8-23所示。

代码清单8-23　Models/ViewModels文件夹下的PagingInfo.cs文件的内容

```
using System;

namespace SportsStore.Models.ViewModels {

    public class PagingInfo {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages =>
            (int)Math.Ceiling((decimal)TotalItems / Ite
msPerPage);
    }
}
```

2．添加标签助手类

现在已经有了视图模型，接下来可以创建标签助手类。在SportsStore项目中创建Infrastructure文件夹，并添加名为PageLinkTagHelper.cs的类文件，该文件用于定义代码清单8-24所示的类。标签助手是

ASP.NET Core MVC的重要组成部分，第23～25章将解释它们的工作原理以及如何创建它们。

代码清单8-24　Infrastructure文件夹下的PageLinkTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model"
)]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory help
erFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]

        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }
```

```
        public string PageAction { get; set; }

        public override void Process(TagHelperContext c
ontext,
                TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.Get
UrlHelper(ViewContext);
            TagBuilder result = new TagBuilder("div");
            for (int i = 1; i <= PageModel.TotalPages;
i++) {
                TagBuilder tag = new TagBuilder("a");
                tag.Attributes["href"] = urlHelper.Acti
on(PageAction,
                   new { productPage = i });
                tag.InnerHtml.Append(i.ToString());
                result.InnerHtml.AppendHtml(tag);
            }
            output.Content.AppendHtml(result.InnerHtml)
;
        }
    }
}
```

提　示

Infrastructure文件夹是放置为应用程序提供基础设施的类的地方，这些类与应用程序的域无关。

标签助手使用与产品页面对应的元素填充div元素。现在不会详细介绍标签助手，知道它们是可以将C#逻辑引入视图的最有用方法之一就足够了。标签助手的代码可能会比较难懂，因为C#和HTML不容易混合。但是使用标签助手要比在视图中包含C#代码块好一些，因为标签助手可以轻松进行单元测试。

大多数MVC组件（如控制器和视图）可以直接使用，但标签助手必须先注册。在代码清单8-25中，在Views文件夹下的_ViewImports.cshtml文件中添加一条语句，告诉MVC在SportsStore.Infrastructure命名空间中查找标签助手

类。这里还添加了一个@using表达式，以便可以引用视图中的视图模型类，而无须使用命名空间限定名称。

代码清单8-25　在Views/Shared文件夹下的ViewImports.cshtml文件中注册一个标签助手

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

单元测试——创建分页链接

为了测试PageLinkTagHelper标签助手类，可使用测试数据调用Process方法，并提供TagHelperOutput对象，以查看生成的HTML，如下所示，在SportsStore项目的PageLinkTagHelperTests.cs文件中定义了这些内

容：

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests {

    public class PageLinkTagHelperTests {

        [Fact]
        public void Can_Generate_Page_Links() {
            // Arrange
            var urlHelper = new Mock<IUrlHelper>();
            urlHelper.SetupSequence(x => x.Action(It.Is
Any<UrlActionContext>()))
                .Returns("Test/Page1")
                .Returns("Test/Page2")
                .Returns("Test/Page3");

            var urlHelperFactory = new Mock<IUrlHelperF
actory>();
            urlHelperFactory.Setup(f =>
                    f.GetUrlHelper(It.IsAny<ActionConte
xt>()))
                        .Returns(urlHelper.Object);

            PageLinkTagHelper helper =
```

```
                       new PageLinkTagHelper(urlHelperF
actory.Object) {
                PageModel = new PagingInfo {
                    CurrentPage = 2,
                    TotalItems = 28,
                    ItemsPerPage = 10
                },
                PageAction = "Test"
            };

            TagHelperContext ctx = new TagHelperContext
(
                new TagHelperAttributeList(),
                new Dictionary<object, object>(), "");
            var content = new Mock<TagHelperContent>();
            TagHelperOutput output = new TagHelperOutpu
t("div",
                new TagHelperAttributeList(),
                (cache, encoder) => Task.FromResult(con
tent.Object));

            // Act
            helper.Process(ctx, output);

            // Assert
            Assert.Equal(@"<a href=""Test/Page1"">1</a>
"
                + @"<a href=""Test/Page2"">2</a>"
                + @"<a href=""Test/Page3"">3</a>",
                 output.Content.GetContent());
        }
    }
}
```

以上测试的复杂性在于创建一些对象，用来创建和使用标签助手。标签助手使用IUrlHelperFactory对象来生成针对应用程序不同部分的URL，并且已经使用Moq来创建对应接口的实现，以及提供测试数据的IUrlHelper接口。

以上测试的核心部分通过使用包含双引号的字符串来验证标签助手的输出。只要字符串的前缀为@，并使用两组双引号代替一组双引号，C#就可以很好地处理这些字符串。记住，不要将字符串分割成单独的行，除非希望拆散正在比较的字符串。例如，在测试方法中使用的字符串已经被换行了，因为打印页面的宽度很窄。这里没有添加换行符；如果这样做，测试将会失败。

---

3. 添加视图模型数据

为了使用标签助手，还需要为视图提供PagingInfo视图模型类的实例。使用view bag功能可以做到这一点，但是作者常把控制器发送的所有数据包装到单个视图模型类的视图中。为此，将一个名为ProductsListViewModel.cs的类文件添加到SportsStore项目的Models/ViewModels文件夹中，代码清单8-26显示了这个文件的内容。

代码清单8-26　Models/ViewModels文件夹下的ProductsListViewModel.cs文件的内容

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    public class ProductsListViewModel {
        public IEnumerable<Product> Products { get; set
; }
        public PagingInfo PagingInfo { get; set; }
    }
}
```

可以更新ProductController类中的List操作方法

来使用ProductsListViewModel类，以提供页面上显示的产品的详细信息以及分页的详细信息，如代码清单8-27所示。

代码清单8-27　更新Controllers文件夹下的ProductController.cs文件中的List操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;
        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }

        public ViewResult List(int productPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
```

```
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Co
unt()
                }
            });
    }
}
```

这些更改会将ProductsListViewModel对象作为模型数据传递给视图。

单元测试——分页模型视图数据

以下是在测试项目中添加到ProductControllerTests类的单元测试，以确保控制器将正确的分页数据发送到视图：

```
...
[Fact]
public void Can_Send_Pagination_View_Model() {

    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
```

```
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    // Arrange
    ProductController controller =
        new ProductController(mock.Object) { PageSize =
 3 };

    // Act
    ProductsListViewModel result =
        controller.List(2).ViewData.Model as ProductsLi
stViewModel;

    // Assert
    PagingInfo pageInfo = result.PagingInfo;
    Assert.Equal(2, pageInfo.CurrentPage);
    Assert.Equal(3, pageInfo.ItemsPerPage);
    Assert.Equal(5, pageInfo.TotalItems);
    Assert.Equal(2, pageInfo.TotalPages);
}
...
```

还需要修改早期的包含在 Can_Paginate 方法中的分页单元测试。它依赖 List 操作方法返回一个 ViewResult，其Model属性是Product对象的序列，

并且需要将数据包含在另一视图模型类型中。下面是修改后的测试：

```
...
[Fact]
public void Can_Paginate() {
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    ProductController controller = new ProductControlle
r(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result =
        controller.List(2).ViewData.Model as ProductsLi
stViewModel;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.True(prodArray.Length == 2);
    Assert.Equal("P4", prodArray[0].Name);
    Assert.Equal("P5", prodArray[1].Name);
}
```

```
...
```

考虑到这两种测试方法之间存在重复，通常会创建一种通用的设置方法。

由于视图目前期望得到Product对象序列，因此需要更新List.cshtml文件，如代码清单8-28所示，以处理新的视图模型类型。

代码清单8-28　更新Views/Product文件夹下的List.cshtml文件

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

这里已经更新了@model指令，以告诉Razor现

在正在使用不同的数据类型。此外，还更新了foreach循环，因此数据源要改为模型数据的Products属性。

4．显示分页链接

现在，将页面链接添加到列表视图所需的代码已经写完，创建了包含分页信息的视图模型，更新了控制器以便将信息传递给视图，还更改了@model指令以匹配新的模型视图类型。剩下的就是添加由标签助手生成的用于创建页面链接的HTML元素，如代码清单8-29所示。

代码清单8-29　在Views/Product文件夹下的List.cshtml文件中添加分页链接

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
```

```
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="List">
</div>
```

如果运行应用程序，你将看到新的页面链接，如图8-9所示。使用的还是基本的样式，这将在本章的后面修正。现在重要的是，新的链接使用户能够翻页探索要销售的产品。当Razor在div元素上找到page-model属性时，要求PageLinkTagHelper类转换元素，进而产生图8-9所示的一组链接。

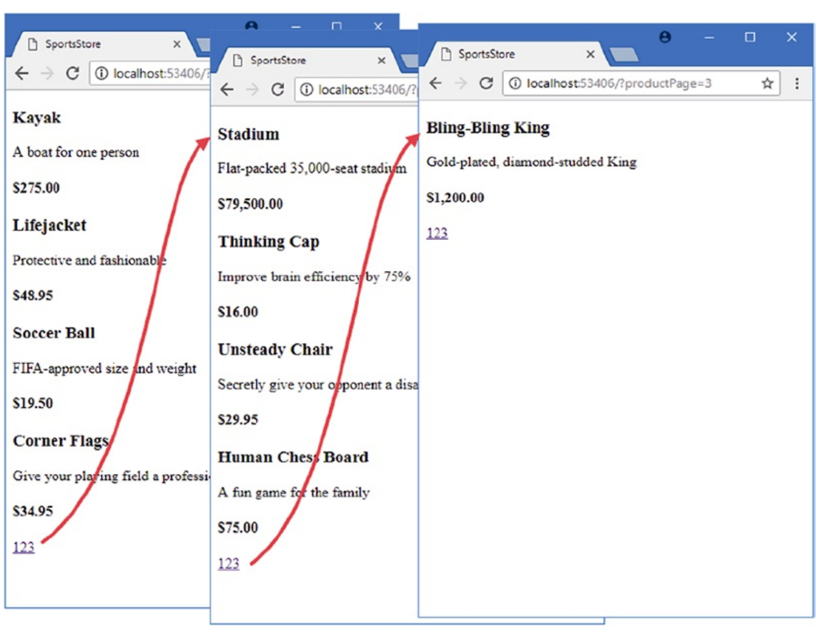


图8-9　页面导航链接

为什么不使用**GridView**？

如果以前曾经使用过ASP.NET，那么可能会认为这是再普通不过的结果。如果使用的是Web窗体，那么可以通过ASP.NET Web Forms的GridView或ListView控件，直接连接到Products数据表，完成相同的操作。

本章完成的内容可能看起来不是很多，但与将控件拖到设计界面上相比有根本的区别。首先，这里正在创建一个具有可靠和可维护性架构的应用程序，涉及关注点分离问题。与简单地使用ListView控件不同，没有直接耦合UI和数据库，虽然这可以快速得到结果，但随着时间的推移会产生很多问题。其次，这里一直在创建单元测试，希望能够以

一种自然的方式验证应用程序的行为，这对于复杂的Web Forms控件几乎是不可能的。最后，请记住，本章已经介绍了很多内容来创建正在构建的应用程序的基础架构。定义和实现存储库即可。例如，现在可以快速、轻松地构建和测试新功能。

当然，正如第3章解释的那样，即使在大型复杂项目中，即时展示数据可能昂贵且很痛苦，但所有这些都不会掩盖如下事实，即Web窗体可以快速地显示结果。

8.5.2　改进URL

现在页面链接可以正常工作，但它们仍然使用查询字符串将页面信息传递到服务器，如下所示：

```
http://localhost/?productPage=2
```

可通过创建一种遵循可组合网址模式的方案，得到更具吸引力的URL。可组合的URL是对用户有意义的URL，就像下面这样：

```
http://localhost/Page2
```

MVC可以轻松地更改应用程序中的URL结构，因为它使用了ASP.NET路由功能，该功能负责处理URL，以确定要访问应用程序的哪一部分。你需要做的是，当你在Startup类的Configure方法中注册MVC中间件时，添加一个新的路由，如代码清单8-30所示。

代码清单8-30　在SportsStore文件夹下的Startup.cs文件中添加一个新的路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));
            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "pagination",
                    template: "Products/Page{productPag
e}",
```

```
                    defaults: new { Controller = "Produ
ct", action = "List" });

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{ac
tion=List}/{id?}");
            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

重要的是，必须将这个路由添加到方法中已有的默认路由之前。如第15章所述，路由系统按照列出的顺序处理路由，新路由需要优先于现有路由。

这就是更改产品分页的URL方案所需的唯一更改。MVC和路由功能是紧密集成的，因此应用程序会自动反映所使用URL中的更改，包括由标签助手生成的URL，以及用于生成页面导航链接的URL。如果现在理解不了路由，不要担心，第15章和第16章将详细解释路由。

如果运行应用程序并单击分页链接，你将看到新的URL结构，如图8-10所示。

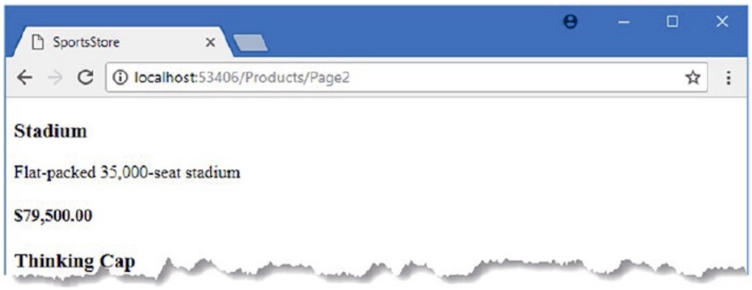


图8-10　浏览器中显示的新的URL结构

8.6　更改内容样式

我们已经创建了大量的底层代码，并且开始集成应用程序的基本功能，但是我们没有在外观上花费太多精力。SportsStore应用程序的设计如此糟糕，即使本书不是关于设计或CSS的，也会影响专业性。本节将修正这些问题，实现带有一个header的经典两列布局，如图8-11所示。



图8-11　SportsStore应用程序的设计目标

8.6.1　安装Bootstrap包

下面使用Bootstrap包来提供将要应用于应用程序的CSS样式。因为需要依赖Visual Studio对Bower的支持来安装Bootstrap包，所以在Add New Item对话框的General类别中选择Bower Configuration File模板，在SportsStore项目中创建名为bower.json的文件，如第6章所示。然后将Bootstrap包添加到创建的bower.json文件的dependencies部分，如代码清单8-31所示。就像之前所说的那样，在本书中为此例使用预发行版的Bootstrap。

代码清单8-31　将Bootstrap添加到SportsStore项目的bower.json

文件中

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

当保存对bower.json文件所做的更改时，Visual Studio使用Bower将Bootstrap软件包下载到wwwroot/lib/ bootstrap文件夹中。Bootstrap依赖于jQuery包，所以jQuery也将自动添加到项目中。

8.6.2 将Bootstrap样式应用于布局

第5章解释了Razor布局如何工作，如何使用它们以及它们如何组成布局。本章开头添加的视图启动文件指定使用名为_Layout.cshtml的文件作为默认布局，这就是应用初始Bootstrap样式的地方，如代码清单8-32所示。

代码清单8-32　将Bootstrap CSS应用于Views/Shared文件夹下的_Layout.cshtml文件

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet"
          asp-href-include="/lib/bootstrap/dist/**/*.mi
n.css"
          asp-href-exclude="**/*-reboot*,**/*-grid*" />
    <title>SportsStore</title>
</head>
<body>
    <div class="navbar navbar-inverse bg-inverse" role=
"navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</
a>
    </div>
    <div class="row m-1 p-1">
        <div id="categories" class="col-3">
            Put something useful here later
        </div>
        <div class="col-9">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

上述代码中的link元素包含asp-href-include和asp-href-exclude属性，展示了一个内置标签助手类的例子。在这种情况下，标签助手将查看属性的值，并生成与指定路径匹配的所有文件的link元素，其中可包含通配符。这个功能很有用，能够确保可以在不破坏应用程序的情况下添加和删除wwwroot文件夹结构中的文件，但是正如第25章解释的那样，需要注意确保指定的路径仅选择期望的文件。

将Bootstrap CSS样式表添加到布局意味着可以在依赖于布局的任何视图中使用它定义样式。在代码清单8-33中，可以看到应用于List.cshtml文件的样式。

代码清单8-33　Views/Product文件夹下的List.cshtml文件中的样式内容

```
@model ProductsListViewModel
```

```
@foreach (var p in Model.Products) {
    <div class="card card-outline-primary m-1 p-1">
        <div class="bg-faded p-1">
            <h4>
                @p.Name
                <span class="badge badge-pill badge-pri
mary" style="float:right">
                    <small>@p.Price.ToString("c")</smal
l>
                </span>
            </h4>
        </div>
        <div class="card-text p-1">@p.Description</div>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="List"
page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-secondary"
    page-class-selected="btn-primary" class="btn-group
pull-right m-1">
</div>
```

需要为PageLinkTagHelper类生成的按钮设置样式，但是这里不会将Bootstrap样式硬编码到C#代码中，因为当在应用程序的其他位置重用标签助手或更改按钮的外观时，这样会更加困难。相反，这里已经在div元素上定义了自定义属性，指定了需要的

样式，它们对应添加到标签助手类中的属性，然后对生成的a元素设置样式，如代码清单8-34所示。

代码清单8-34　将样式添加到PageLinkTagHelper.cs文件中生成的元素上

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model"
)]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory help
erFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }
```

```
        public string PageAction { get; set; }

        public bool PageClassesEnabled { get; set; } =
false;
        public string PageClass { get; set; }
        public string PageClassNormal { get; set; }
        public string PageClassSelected { get; set; }
        public override void Process(TagHelperContext c
ontext,
                TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.Get
UrlHelper(ViewContext);
            TagBuilder result = new TagBuilder("div");
            for (int i = 1; i <= PageModel.TotalPages;
i++) {
                TagBuilder tag = new TagBuilder("a");
                tag.Attributes["href"] = urlHelper.Acti
on(PageAction,
                    new { productPage = i });
                if (PageClassesEnabled) {
                    tag.AddCssClass(PageClass);
                    tag.AddCssClass(i == PageModel.Curr
entPage
                        ? PageClassSelected : PageClass
Normal);
                }
                tag.InnerHtml.Append(i.ToString());
                result.InnerHtml.AppendHtml(tag);
            }
            output.Content.AppendHtml(result.InnerHtml)
;
        }
    }
}
```

这些属性的值将自动用于设置标签助手的属性值，并考虑HTML属性名称格式（page-class-normal）和C#属性名称格式（PageClassNormal）之间的映射。这允许标签助手根据HTML元素的属性进行不同的响应，从而创建一种更灵活的方法在MVC应用程序中生成内容。

如果运行应用程序，你将看到应用程序的外观已经得到一些改进，如图8-12所示。

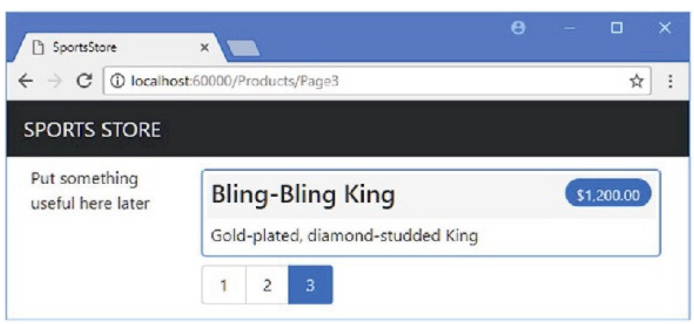


图8-12　外观得到改进的SportsStore应用程序

8.6.3　创建分部视图

作为本章的结尾，下面重构应用程序以简化

List.cshtml视图文件。创建一个分部视图（partial view），它是一个内容片段，可以嵌入另一个视图，有点类似于模板。第21章将详细描述分部视图，并且当你需要在应用程序的不同位置显示相同的内容时，它们有助于减少重复。可以在分部视图中定义一次，而不是将相同的Razor标记复制并粘贴到多个视图中。为了创建分部视图，在Views/Shared文件夹添加名为ProductSummary.cshtml的Razor视图文件，并添加代码清单8-35所示的代码。

代码清单8-35　Views/Shared文件夹下的ProductSummary.cshtml文件中的内容

```
@model Product

<div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
        <h4>
            @Model.Name
            <span class="badge badge-pill badge-primary
" style="float:right">
                <small>@Model.Price.ToString("c")</smal
```

```
l>
            </span>
        </h4>
    </div>
    <div class="card-text p-1">@Model.Description</div>
</div>
```

现在需要更新Views/Products文件夹中的List.cshtml文件，以便使用分部视图，如代码清单8-36所示。

代码清单8-36　在List.cshtml文件中使用分部视图

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div page-model="@Model.PagingInfo" page-action="List"
page-classes-enabled="true"
     page-class="btn" page-class-normal="btn-secondary"
     page-class-selected="btn-primary" class="btn-group
 pull-right m-1">
</div>
```

这里已经将List.cshtml文件中的foreach循环中的标记移动到新的分部视图。使用Html.Partial辅助

方法调用分部视图，其中包含视图名称和视图模型对象的参数。切换到这样的分部视图是比较好的做法，从而允许将相同的标记插入需要显示产品摘要的任何视图中。如图8-13所示，添加分部视图不会改变应用程序的外观；只是改变了Razor搜索生成内容的位置，从而生成发送到浏览器的响应。

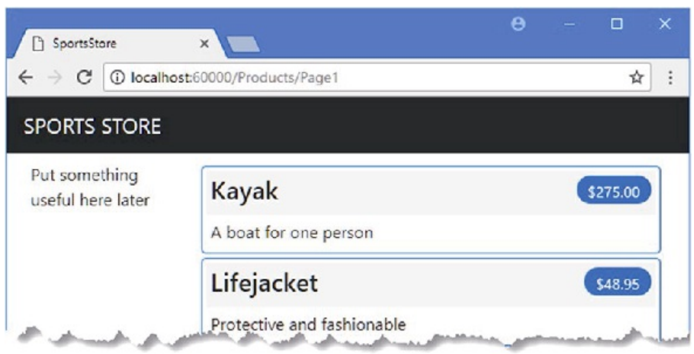


图8-13　应用分部视图

8.7　小结

本章为SportsStore应用程序构建了核心基础架构。现在还没有很多功能可供你向客户展示，但我们已经开始了领域模型开发，其产品存储库由SQL

Server和Entity Framework Core支持。另一个控制器ProductController可以生成分页的产品列表。我们还创建了一种简洁、友好的URL结构。

现在底层的基础设施代码已经到位，我们可以继续增加面向客户的所有功能——按类别导航、购物车以及结账。

第9章　SportsStore的导航

本章将继续构建SportsStore应用程序，添加对应用程序导航的支持并开始构建购物车。

9.1　添加导航控件

如果客户可以按类别浏览产品，SportsStore应用程序将更为方便，这可分3个阶段完成。

- 增强ProductController类中的List操作模型，以便能够过滤存储库中的Product对象。
- 重构并增强URL结构。
- 创建类别列表，展示在站点的边栏中，突出显示当前类别并链接到其他类别。

9.1.1　过滤产品列表

为了改进视图模型类（上一章添加到SportsStore项目中的ProductsListViewModel），需要让当前类别与视图进行通信，以便渲染边栏，这是一个很好的起点。代码清单9-1显示了对Models/ViewModels文件夹中的ProductsListViewModel.cs文件所做的更改。

代码清单9-1　在Models/ViewModels文件夹下的ProductsListViewModel.cs文件中添加一个属性

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    public class ProductsListViewModel {
        public IEnumerable<Product> Products { get; set
; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}
```

这里添加了一个名为CurrentCategory的属性。

下一步是更新Product控制器，以便List操作方法按类别过滤Product对象，并使用添加到视图模型中的新属性来指示哪个类别被选中。代码清单9-2显示了所做的更改。

代码清单9-2　在Controllers文件夹下的ProductController.cs文件的List操作方法中添加对类别的支持

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }

        public ViewResult List(string category, int pro
ductPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.C
```

```
ategory == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Co
unt()
                },
                CurrentCategory = category
            });
    }
}
```

这里对这个操作方法做了3处更改。首先，添加一个名为category的参数，category参数由列表中的第二处更改使用，这是LINQ查询的增强功能：如果类别不为空，那么仅选择匹配Category属性的Product对象。最后一处更改是设置添加到ProductsListViewModel类的CurrentCategory属性的值。但是，这些更改意味着没有正确计算PagingInfo.TotalItems的值，因为没有考虑类别过滤器，稍后再解决这个问题。

单元测试——更新已存在的单元测试

这里更改了List操作方法的签名，这将影响编译一些现有的单元测试方法。为了解决这个问题，需要将null作为第一个参数传递给使用控制器的单元测试中的List方法。例如，在ProductControllerTests.cs文件的Can_Paginate测试中，单元测试的操作部分如下：

```
...
[Fact]
public void Can_Paginate() {
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());
```

```
    ProductController controller = new ProductControlle
r(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result =
        controller.List(null, 2).ViewData.Model as Prod
uctsListViewModel;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.True(prodArray.Length == 2);
    Assert.Equal("P4", prodArray[0].Name);
    Assert.Equal("P5", prodArray[1].Name);
}
...
```

通过对category参数使用null，就能得到控制器从存储库中获取的所有Product对象，这与添加新参数之前的情况相同。此外，还需要对Can_Send_Pagination_View_Model测试做同样的修改。

```
...
[Fact]
public void Can_Send_Pagination_View_Model() {
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
```

```
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    // Arrange
    ProductController controller =
        new ProductController(mock.Object) { PageSize =
 3 };

    // Act
    ProductsListViewModel result =
        controller.List(null, 2).ViewData.Model as Prod
uctsListViewModel;

    // Assert
    PagingInfo pageInfo = result.PagingInfo;
    Assert.Equal(2, pageInfo.CurrentPage);
    Assert.Equal(3, pageInfo.ItemsPerPage);
    Assert.Equal(5, pageInfo.TotalItems);
    Assert.Equal(2, pageInfo.TotalPages);
}
...
```

当进入测试模式时，使单元测试与代码更改快速同步成为第二条重要的原则。

要查看类别过滤的效果，请启动应用程序并使用以下查询字符串选择一个类别，将端口更改为Visual Studio为项目分配的端口（注意，Soccer的首字母要大写）：

```
http://localhost:60000/?category=Soccer
```

你将只能看到足球类别中的商品，如图9-1所示。

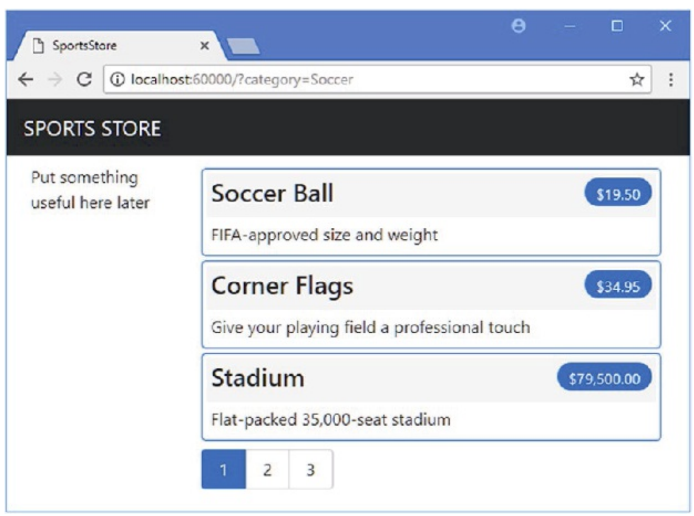


图9-1　使用查询字符串按类别过滤商品

显然，用户不会使用URL导航到类别，但是一

旦基础设施代码到位，就可以看到MVC应用程序中一个小的更改也可以产生很大的影响。

单元测试——类别过滤

你需要使用单元测试来测试类别过滤功能，以确保过滤器能够正确地生成指定类别的商品。下面是添加到ProductControllerTests类的测试方法：

```
...
[Fact]
public void Can_Filter_Products() {

    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1", Catego
ry = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Catego
ry = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Catego
ry = "Cat1"},
```

```
        new Product {ProductID = 4, Name = "P4", Catego
ry = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Catego
ry = "Cat3"}
    }).AsQueryable<Product>());

    // Arrange - create a controller and make the page
size 3 items
    ProductController controller = new ProductControlle
r(mock.Object);
    controller.PageSize = 3;

    // Action
    Product[] result =
        (controller.List("Cat2", 1).ViewData.Model as P
roductsListViewModel)
            .Products.ToArray();

    // Assert
    Assert.Equal(2, result.Length);
    Assert.True(result[0].Name == "P2" && result[0].Cat
egory == "Cat2");
    Assert.True(result[1].Name == "P4" && result[1].Cat
egory == "Cat2");
}
...
```

以上测试将创建一个模拟的存储库，其中包含属于多个类别的Product对象。使用一个操作方法请求某个特定类别，并检查结果以确保结果是正确的对象并具有正确的顺序。

9.1.2　优化URL结构

没人想看到或使用丑陋的网址，如/?category = Soccer。为了解决这个问题，可在Startup类的Configure方法中更改路由配置，以创建一组更友好的URL，如代码清单9-3所示。

代码清单9-3　更改SportsStore文件夹下的Startup.cs文件中的路由方案

```
...
public void Configure(IApplicationBuilder app, IHosting
Environment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvc(routes => {

        routes.MapRoute(
            name: null,
            template: "{category}/Page{productPage:int}
",
            defaults: new { controller = "Product", act
ion = "List" }
        );
```

```
        routes.MapRoute(
            name: null,
            template: "Page{productPage:int}",
            defaults: new { controller = "Product",
                action = "List", productPage = 1 }
        );

        routes.MapRoute(
            name: null,
            template: "{category}",
            defaults: new { controller = "Product",
                action = "List", productPage = 1 }
        );

        routes.MapRoute(
            name: null,
            template: "",
            defaults: new { controller = "Product", act
ion = "List",
                productPage = 1 });

        routes.MapRoute(name: null, template: "{control
ler}/{action}/{id?}");
    });
    SeedData.EnsurePopulated(app);
}
...
```



警　告

按照代码清单9-3中的顺序添加新路由非常重要。路由按定义的顺序生效，如果更改顺序，你会得到一些奇怪的结果。

这些路由表示的URL结构如表9-1所示。第15章和第16章将详细解释路由系统。

表**9-1**　路由摘要

URL	导航到
/	列出所有类别商品的第一页
/Page2	列出特定页面（在本例中为第2页），显示所有类别的商品
/Soccer	显示来自特定类别的商品的第一页（在本例中为Soccer类别）
/Soccer/Page2	显示指定类别（在这种情况下为Soccer）的商品的指定页面（在本例中为第2页）

MVC使用ASP.NET Core路由系统来处理来自客户端的传入请求，并且生成符合URL结构的可嵌入网页中的传出URL。通过使用路由系统来处理传入请求和传出URL，可以确保应用程序中的所有URL一致。

IUrlHelper接口提供对URL生成功能的访问。你在上一章中创建的标签助手使用了这个接口，以及其中定义的操作方法。现在我们想开始生成更复杂的URL，我们需要一种从视图中接收附加信息的方法，而不必为标签助手类添加额外的属性。幸运的是，标签助手有一个很好的功能，允许属性拥有公共前缀，从而可以在单个集合中接收，如代码清单9-4所示。

代码清单9-4　在Infrastructure文件夹下的PageLinkTagHelper.cs文件中接收前缀属性值

```
using Microsoft.AspNetCore.Mvc;
```

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
using System.Collections.Generic;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model"
)]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory help
erFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        [HtmlAttributeName(DictionaryAttributePrefix =
"page-url-")]
        public Dictionary<string, object> PageUrlValues
 { get; set; }
            = new Dictionary<string, object>();

        public bool PageClassesEnabled { get; set; } =
false;
        public string PageClass { get; set; }
```

```
        public string PageClassNormal { get; set; }
        public string PageClassSelected { get; set; }

        public override void Process(TagHelperContext c
ontext,
                TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.Get
UrlHelper(ViewContext);
            TagBuilder result = new TagBuilder("div");
            for (int i = 1; i <= PageModel.TotalPages;
i++) {
                TagBuilder tag = new TagBuilder("a");
                PageUrlValues["productPage"] = i;
                tag.Attributes["href"] = urlHelper.Acti
on(PageAction, PageUrlValues);
                if (PageClassesEnabled) {
                    tag.AddCssClass(PageClass);
                    tag.AddCssClass(i == PageModel.Curr
entPage
                        ? PageClassSelected : PageClass
Normal);
                }
                tag.InnerHtml.Append(i.ToString());
                result.InnerHtml.AppendHtml(tag);
            }
            output.Content.AppendHtml(result.InnerHtml)
;
        }
    }
}
```

通过使用HtmlAttributeName特性（attribute）装饰一个标签助手属性（property），可以为元素

上的属性名称指定前缀，在这个例子中是page-url-。名称以此前缀开头的任何属性的值都将被添加到分配给PageUrlValues属性的字典中，然后传递给IUrlHelper.Action方法，以生成由标签助手输出的a元素的href属性。

在代码清单9-5中，已经为标签助手处理的div元素添加了一个新的属性，指定了用于生成URL的类别。虽然只为视图添加了一个新的属性，但是具有相同前缀的任何属性都将被添加到字典中。

代码清单9-5　在Views/Home文件夹下的List.cshtml文件中添加新属性

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div page-model="@Model.PagingInfo" page-action="List"
page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-secondary"
    page-class-selected="btn-primary" page-url-category
```

```
="@Model.CurrentCategory"
    class="btn-group pull-right m-1">
</div>
```

在进行上述更改之前，为分页链接生成的链接如下所示：

```
http://<myserver>:<port>/Page1
```

如果用户单击这样的页面链接，类别过滤器将会丢失，应用程序将显示一个包含所有类别产品的页面。通过添加从视图模型中获取的当前类别，就会生成下面这样的URL：

```
http://<myserver>:<port>/Chess/Page1
```

当用户单击此类链接时，当前类别将被传递给List操作方法，并且会保留过滤功能。进行此更改后，可以访问/Chess或/Soccer等网址，你会看到页面底部的页面链接已正确包含该类别。

9.1.3　构建类别导航菜单

你需要为客户提供一种方式，无须用户在URL中输入类别。这意味着需要向他们呈现可用类别的列表，并指出当前选择了哪个类别（如果有的话）。当构建应用程序时，将在多个控制器中使用这个类别列表，所以需要一些自包含且可重用的东西。

ASP.NET Core MVC具有视图组件的概念，这对于创建诸如可重复使用的导航控件之类的项目是非常合适的。视图组件是C#类，可以提供少量可重用的应用程序逻辑，能够选择和显示Razor分部视图。第22章将详细描述视图组件。

在本节的例子中，将创建一个视图组件，通过从共享布局调用组件来呈现导航菜单，并将其集成到应用程序中。这种方法让我们能够使用一个常规

的C#类，可以包含我们所需要的任何应用程序逻辑，并且可以像任何其他类一样进行单元测试。这是一种在保留整体MVC方法的同时创建应用程序的较小片段的好方法。

1．创建导航视图组件

创建名为Components的文件夹（一般用来存放视图组件），并在其中添加名为NavigationMenuViewComponent.cs的类，用于定义代码清单9-6所示的类。

代码清单9-6　Components文件夹下的NavigationMenuViewComponent.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComp
onent {

        public string Invoke() {
```

```
            return "Hello from the Nav View Component";
        }
    }
}
```

在Razor视图中使用组件时，将调用视图组件的Invoke方法，并将Invoke方法的结果插入HTML中，发送到浏览器。我们已经开始使用一个返回字符串的简单视图组件，但我们很快就会用动态HTML内容替换它。

这里希望类别列表出现在所有页面上，因此我们将在共享布局中使用视图组件，而不是在特定视图中。在视图中，可通过@await Component.InvokeAsync表达式使用视图组件，如代码清单9-7所示。

代码清单9-7　在Views/Shared文件夹下的_Layout.cshtml文件中使用视图组件

```
<!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet"
          asp-href-include="/lib/bootstrap/dist/**/*.mi
n.css"
          asp-href-exclude="**/*-reboot*,**/*-grid*" />
    <title>SportsStore</title>
</head>
<body>
    <div class="navbar navbar-inverse bg-inverse" role=
"navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</
a>
    </div>
    <div class="row m-1 p-1">
        <div id="categories" class="col-3">
            @await Component.InvokeAsync("NavigationMen
u")
        </div>
        <div class="col-9">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

这里删除了占位符文本，并将其替换为对Component.InvokeAsync方法的调用。此方法的参数是组件类的名称，省略了类名的ViewComponent部

分，因此NavigationMenu指定了NavigationMenuViewComponent类。如果运行该应用程序，你将看到Invoke方法的输出包含在发送给浏览器的HTML中，如图9-2所示。

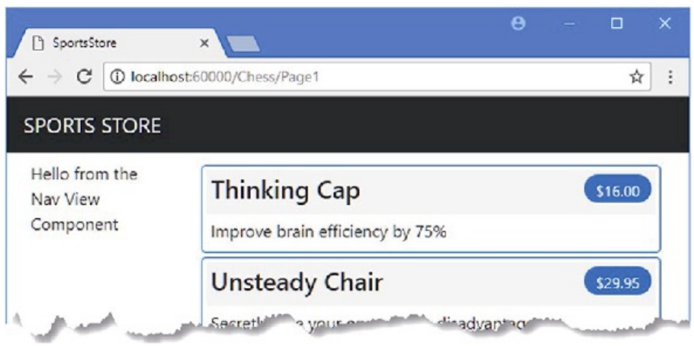


图9-2　使用视图组件的输出

2．生成类别列表

现在可以返回导航视图控制器并生成一组真实的类别。可以使用编程方式为类别构建HTML，就像为页面标签助手所做的那样，但使用视图组件的一个好处是它们可以渲染Razor分部视图。这意味着可以使用视图组件生成组件列表，然后使用更具

表现力的Razor语法来呈现用于显示它们的HTML。第一步是更新视图组件，如代码清单9-8所示。

代码清单9-8　在Components文件夹下的NavigationMenuViewComponent.cs文件中添加类别

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComp
onent {
        private IProductRepository repository;

        public NavigationMenuViewComponent(IProductRepo
sitory repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

代码清单9-8中的构造函数定义了一个IProductRepository参数。当MVC需要创建视图组件类的实例时，它将需要提供这个参数并检查Startup类中的配置以确定应使用哪个实现对象。这与第8章在控制器中使用的依赖注入功能相同，并且具有相同的效果，即允许视图组件在不知道将使用哪个存储库实现的情况下访问数据，如第18章所述。

在Invoke方法中，使用LINQ对存储库中的一组类别进行选择和排序，并将它们作为参数传递给View方法，View方法会呈现默认的Razor分部视图，使用IViewComponentResult对象从方法返回其详细信息，第22章会详细描述此过程。

单元测试——生成类别列表

用于类别列表的单元测试相对简单。目标是创

建一个按字母顺序排列的列表，并且不包含重复项。最简单的方法是提供一些具有重复类别且不按顺序排列的测试数据，将其传递给标签助手类，并判断数据是否已被正确清理。在SportsStore.Tests项目中，在名为NavigationMenuViewComponentTests.cs的类文件中定义的单元测试如下：

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using SportsStore.Components;
using SportsStore.Models;
using Xunit;
namespace SportsStore.Tests {

    public class NavigationMenuViewComponentTests {

        [Fact]
        public void Can_Select_Categories() {
            // Arrange
            Mock<IProductRepository> mock = new Mock<IP
roductRepository>();
            mock.Setup(m => m.Products).Returns((new Pr
oduct[] {
                new Product {ProductID = 1, Name = "P1"
```

```
, Category = "Apples"},
                new Product {ProductID = 2, Name = "P2"
, Category = "Apples"},
                new Product {ProductID = 3, Name = "P3"
, Category = "Plums"},
                new Product {ProductID = 4, Name = "P4"
, Category = "Oranges"},
            }).AsQueryable<Product>());

            NavigationMenuViewComponent target =
                new NavigationMenuViewComponent(mock.Ob
ject);

            // Act = get the set of categories
            string[] results = ((IEnumerable<string>)(t
arget.Invoke()
                as ViewViewComponentResult).ViewData.Mo
del).ToArray();

            // Assert
            Assert.True(Enumerable.SequenceEqual(new st
ring[] { "Apples",
                "Oranges", "Plums" }, results));
        }
    }
}
```

以上代码创建了一个模拟的存储库实现，其中包含重复的以及没有正确排序的类别。我们判断重复项将被删除，并强制按字母顺序排列。

3．创建视图

Razor使用不同的约定来处理视图组件选择的视图。视图的默认名称和搜索位置与用于控制器的不同。为此，创建Views/Shared/Components/NavigationMenu文件夹，并在其中添加名为Default.cshtml的视图文件，内容如代码清单9-9所示。

代码清单9-9　Views/Shared/Components/NavigationMenu文件夹下的Default.cshtml文件的内容

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
    Home
</a>
@foreach (string category in Model) {
    <a class="btn btn-block btn-secondary"
```

```
        asp-action="List"
        asp-controller="Product"
        asp-route-category="@category"
        asp-route-productPage="1">
         @category
    </a>
}
```

这个视图使用第24章和第25章描述的内置标签助手之一来创建a元素，其href属性包含选择不同产品类别的URL。

如果运行应用程序，就可以看到类别链接，如图9-3所示。如果单击某个类别，就会更新条目列表，以仅显示所选类别中的条目。

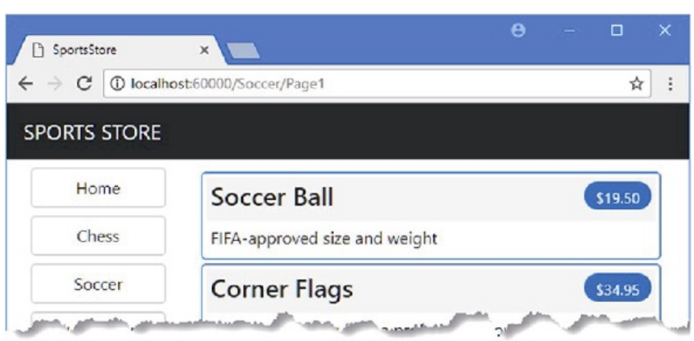


图9-3　使用视图组件生成类别链接

4．突出显示当前类别

现在还没有给用户发出反馈以指示选择了哪个类别。从列表中的条目推断出类别也是可能的，但是提供一些清晰的视觉反馈似乎更好。ASP.NET Core MVC组件（如控制器和视图组件）可以通过请求上下文对象来接收有关当前请求的信息。大多数情况下，可以依赖用于创建组件的基类来获取上下文对象，例如在使用Controller基类创建控制器时。

ViewComponent基类也不例外，它通过一组属性提供对上下文对象的访问。其中一个属性名为RouteData，它提供有关路由系统如何处理请求URL的信息。

代码清单9-10使用RouteData属性来访问请求数据，以获取当前所选类别的值。可以通过创建另一

个视图模型类将类别传递给视图（这就是在实际项目中要做的事情），但是为了展示更多方式，我们将使用第2章介绍的view bag功能。

代码清单9-10　在NavigationMenuViewComponent.cs文件中传递所选类别

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComp
onent {
        private IProductRepository repository;

        public NavigationMenuViewComponent(IProductRepo
sitory repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            ViewBag.SelectedCategory = RouteData?.Value
s["category"];
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
```

```
    }
}
```

在Invoke方法中，已经为ViewBag对象动态分配了SelectedCategory属性，并将属性值设置为当前类别，当前类别是通过RouteData属性返回的上下文对象获得的。正如第2章解释的那样，ViewBag是动态对象，可以简单地为它们分配值来定义新的属性。

单元测试——报告所选类别

可以通过在单元测试中读取ViewBag属性的值来测试视图组件是否正确添加了所选类别的详细信息，该值可通过第22章描述的ViewViewComponentResult类获得。下面是添加到NavigationMenuViewComponentTests类中的单元测

试:

```
...
[Fact]
public void Indicates_Selected_Category() {

    // Arrange
    string categoryToSelect = "Apples";
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1", Catego
ry = "Apples"},
        new Product {ProductID = 4, Name = "P2", Catego
ry = "Oranges"},
    }).AsQueryable<Product>());
    NavigationMenuViewComponent target =
        new NavigationMenuViewComponent(mock.Object);
    target.ViewComponentContext = new ViewComponentCont
ext {
        ViewContext = new ViewContext {
            RouteData = new RouteData()
        }
    };
    target.RouteData.Values["category"] = categoryToSel
ect;

    // Action
    string result = (string)(target.Invoke() as
        ViewViewComponentResult).ViewData["SelectedCate
gory"];

    // Assert
```

```
    Assert.Equal(categoryToSelect, result);
}
...
```

以上单元测试通过ViewComponentContext属性为视图组件提供路由数据，该属性指定了视图组件接收所有上下文数据的方式。ViewComponentContext属性通过ViewContext子属性提供对特定于视图的上下文数据的访问，而后通过RouteData子属性提供对路由信息的访问。单元测试中的大多数代码用于创建上下文对象，这些对象将以与应用程序运行时呈现的相同方式提供所选类别，其上下文数据由ASP.Net Core MVC提供。

现在提供了有关选择哪个类别的信息，可以利用这一点更新视图组件选择的视图，并改变用于设置链接样式的CSS类，使代表当前类别的CSS类与其他类别不同。代码清单9-11显示了对

Default.cshtml文件所做的更改。

代码清单9-11　在Views/Shared/Components/NavigationMenu文件夹下的Default.cshtml文件中突出显示当前类别

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
    Home
</a>

@foreach (string category in Model) {
    <a class="btn btn-block
       @(category == ViewBag.SelectedCategory ? "btn-pr
imary": "btn-secondary")"
       asp-action="List"
       asp-controller="Product"
       asp-route-category="@category"
       asp-route-productPage="1">
        @category
    </a>
}
```

在class属性中使用了一个Razor表达式，从而将btn-primary类应用于表示所选类别的元素，其他类别的元素使用的是btn-secondary类。为这些类别

应用不同的Bootstrap样式，并使活动按钮突出显示，如图9-4所示。

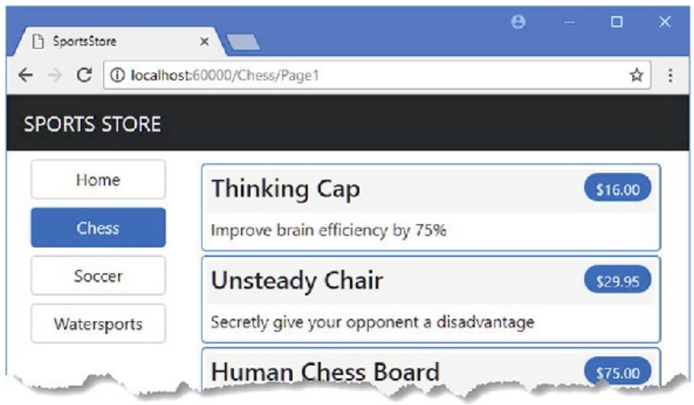


图9-4　突出显示所选类别

9.1.4　更正页数

需要更正页面链接，以便在选择类别时它们能够正常工作。目前，页面链接的数量由存储库中的商品总数确定，而不是由所选类别中的商品数量确定。这意味着客户可以单击Chess类别第2页的链接，但会得到一个空白页面，因为没有足够的Chess商品来填充两个页面，如图9-5所示。

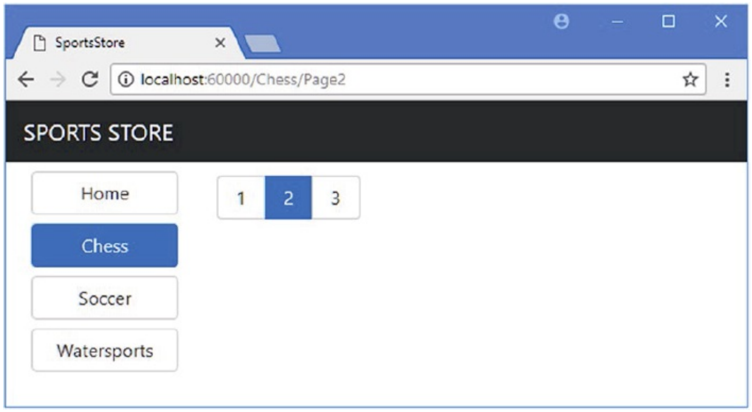


图9-5　选择类别时显示错误的页面链接

可以通过更新Product控制器中的List操作方法来解决这个问题，让分页信息按类别计算，如代码清单9-12所示。

代码清单9-12　在Controllers文件夹下的ProductController.cs文件中创建支持类别的分页数据

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

```

```
        public ProductController(IProductRepository rep
o) {
            repository = repo;
        }

        public ViewResult List(string category, int pro
ductPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.C
ategory == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                    PagingInfo = new PagingInfo {
                        CurrentPage = productPage,
                        ItemsPerPage = PageSize,
                        TotalItems = category == null ?
                            repository.Products.Count()
 :
                            repository.Products.Where(e
 =>
                                e.Category == category)
.Count()
                    },
                    CurrentCategory = category
                });
    }
}
```

如果已选择某个类别，就返回该类别中的商品数量；如果没有，就返回商品总数。现在，当查看

某个类别时，页面底部的链接正确反映了该类别中的商品数量，如图9-6所示。

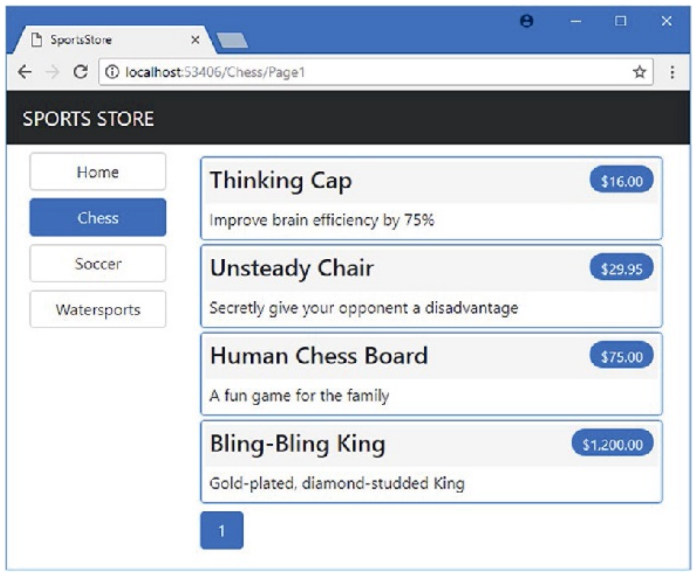


图9-6　显示特定类别页数

测试特定类别的商品计数

为不同类别计算当前商品数量是很简单的。创建一个模拟的存储库，其中包含一系列类别中的已知数据，然后调用List操作方法依次请求每个类

别。以下是添加到ProductControllerTests类中的单元测试方法：

```
...
[Fact]
public void Generate_Category_Specific_Product_Count()
{
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns((new Product[]
{
        new Product {ProductID = 1, Name = "P1", Catego
ry = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Catego
ry = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Catego
ry = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Catego
ry = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Catego
ry = "Cat3"}
    }).AsQueryable<Product>());
    ProductController target = new ProductController(mo
ck.Object);
    target.PageSize = 3;

    Func<ViewResult, ProductsListViewModel> GetModel =
result =>
        result?.ViewData?.Model as ProductsListViewMode
l;

    // Action
```

```
    int? res1 = GetModel(target.List("Cat1"))?.PagingIn
fo.TotalItems;
    int? res2 = GetModel(target.List("Cat2"))?.PagingIn
fo.TotalItems;
    int? res3 = GetModel(target.List("Cat3"))?.PagingIn
fo.TotalItems;
    int? resAll = GetModel(target.List(null))?.PagingIn
fo.TotalItems;

    // Assert
    Assert.Equal(2, res1);
    Assert.Equal(2, res2);
    Assert.Equal(1, res3);
    Assert.Equal(5, resAll);
}
...
```

请注意，以上代码在没有指定类别的情况下调用了List操作方法，以确保得到正确的总数。

9.2 构建购物车

现在应用程序进展顺利，但在实现购物车之前，无法销售任何商品。图9-7展示了基本的购物流程。在网上购物的人都会熟悉这一流程。



图9-7　基本的购物流程

目录中每个产品的旁边都会显示Add to Cart按钮，单击此按钮将显示客户目前所选商品的摘要，包括总价。此时，用户可以单击Continue Shopping按钮返回商品目录，或单击Checkout Now按钮完成订单并结束购物流程。

9.2.1　定义购物车模型

将一个名为Cart.cs的类文件添加到Models文件夹中，并用它定义代码清单9-13所示的类。

代码清单9-13　Models文件夹下的Cart.cs文件的内容

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {
```

```
    public class Cart {
        private List<CartLine> lineCollection = new Lis
t<CartLine>();

        public virtual void AddItem(Product product, in
t quantity) {
            CartLine line = lineCollection
                .Where(p => p.Product.ProductID == prod
uct.ProductID)
                .FirstOrDefault();

            if (line == null) {
                lineCollection.Add(new CartLine {
                    Product = product,
                    Quantity = quantity
                });
            } else {
                line.Quantity += quantity;
            }
        }

        public virtual void RemoveLine(Product product)
 =>
            lineCollection.RemoveAll(l => l.Product.Pro
ductID == product.ProductID);

        public virtual decimal ComputeTotalValue() =>
            lineCollection.Sum(e => e.Product.Price * e
.Quantity);

        public virtual void Clear() => lineCollection.C
lear();

        public virtual IEnumerable<CartLine> Lines => l
ineCollection;
```

```
    }

    public class CartLine {
        public int CartLineID { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

在以上代码中，Cart类使用在同一文件中定义的CartLine类来表示客户选择的商品和用户想要购买的数量，还定义了以下功能：将商品添加到购物车，从购物车中删除以前添加的商品，计算购物车中商品的总价格，以及通过删除所有商品重置购物车。另外，还添加了一个属性，让你可以使用IEnumerable<CartLine>访问购物车的内容。这些内容比较简单，使用C#的LINQ可以轻松实现。

单元测试——测试购物车

Cart 类相对简单，但它有一系列重要的行为，

必须能够正常工作。功能糟糕的购物车会破坏整个SportsStore应用程序。我们已经分解了这些功能并单独测试了它们。在SportsStore.Tests项目中创建一个名为CartTests.cs的单元测试文件，其中包含这些测试。

第一个行为涉及何时将一个商品添加到购物车。如果这是第一次将给定的商品添加到购物车，那么需要添加一个新的CartLine对象。以下是包括单元测试类定义的测试：

```
using System.Linq;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class CartTests {
        [Fact]
        public void Can_Add_New_Lines() {

            // Arrange - create some test products
            Product p1 = new Product { ProductID = 1, N
ame = "P1" };
            Product p2 = new Product { ProductID = 2, N
```

```
ame = "P2" };

            // Arrange - create a new cart
            Cart target = new Cart();

            // Act
            target.AddItem(p1, 1);
            target.AddItem(p2, 1);
            CartLine[] results = target.Lines.ToArray()
;


            // Assert
            Assert.Equal(2, results.Length);
            Assert.Equal(p1, results[0].Product);
            Assert.Equal(p2, results[1].Product);
        }
    }
}
```

但是，如果客户已将商品添加到购物车，则需要增加相应的CartLine对象的数量，而不是创建一个新的。测试如下：

```
...
[Fact]
public void Can_Add_Quantity_For_Existing_Lines() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P
1" };
    Product p2 = new Product { ProductID = 2, Name = "P
2" };
```

```
    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 10);
    CartLine[] results = target.Lines
        .OrderBy(c => c.Product.ProductID).ToArray();

    // Assert
    Assert.Equal(2, results.Length);
    Assert.Equal(11, results[0].Quantity);
    Assert.Equal(1, results[1].Quantity);
}
...
```

另外，还需要检查客户是否可以改变主意，并从购物车中删除商品。该功能由RemoveLine方法实现。测试如下：

```
...
[Fact]
public void Can_Remove_Line() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P
1" };
    Product p2 = new Product { ProductID = 2, Name = "P
2" };
    Product p3 = new Product { ProductID = 3, Name = "P
```

```
3" };

    // Arrange - create a new cart
    Cart target = new Cart();
    // Arrange - add some products to the cart
    target.AddItem(p1, 1);
    target.AddItem(p2, 3);
    target.AddItem(p3, 5);
    target.AddItem(p2, 1);

    // Act
    target.RemoveLine(p2);

    // Assert
    Assert.Equal(0, target.Lines.Where(c => c.Product =
= p2).Count());
    Assert.Equal(2, target.Lines.Count());
}
...
```

这里想要测试的下一个行为是计算购物车中商品的总价。以下是对此行为的测试：

```
...
[Fact]
public void Calculate_Cart_Total() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P
1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P
2", Price = 50M };
```

```
    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 3);
    decimal result = target.ComputeTotalValue();

    // Assert
    Assert.Equal(450M, result);
}
...
```

最后的测试很简单。确保在重置购物车时正确删除了购物车的内容。测试如下：

```
...
[Fact]
public void Can_Clear_Contents() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P
1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P
2", Price = 50M };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Arrange - add some items
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
```

```
    // Act - reset the cart
    target.Clear();

    // Assert
    Assert.Equal(0, target.Lines.Count());
}
...
```

在本例中，有时测试类的功能所需的代码相比类本身更多、更复杂。不要因为复杂就放弃单元测试。简单类中的缺陷可能会产生巨大的影响，特别是在示例应用程序中扮演重要作用的一些类，比如Cart类。

9.2.2 添加Add To Cart按钮

这里需要编辑Views/Shared/ProductSummary.cshtml分部视图以将按钮添加到商品列表中。为准备此功能，这里将一个名为UrlExtensions.cs的类文件添加到Infrastructure

文件夹中，并定义了扩展方法，如代码清单9-14所示。

代码清单9-14　Infrastructure文件夹下的UrlExtensions.cs文件的内容

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

    public static class UrlExtensions {

        public static string PathAndQuery(this HttpRequ
est request) =>
            request.QueryString.HasValue
                ? $"{request.Path}{request.QueryString}
"
                : request.Path.ToString();
    }
}
```

PathAndQuery扩展方法在HttpRequest类上执行操作，ASP.NET则使用HttpRequest类来描述HTTP请求。该扩展方法会生成一个URL，浏览器将在购物车更新后返回，并将查询字符串考虑在内（如果

有的话）。在代码清单9-15中，将包含扩展方法的命名空间添加到视图导入文件中，以便在分部视图中使用它。

代码清单9-15　在Views文件夹下的_ViewImports.cshtml文件中添加命名空间

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

在代码清单9-16中，已经更新了描述每个商品的分部视图，以包含Add To Cart按钮。

代码清单9-16　将按钮添加到Views/Shared文件夹下的ProductSummary.cshtml文件中

```
@model Product

<div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
        <h4>
            @Model.Name
            <span class="badge badge-pill badge-primary
```

```
" style="float:right">
                <small>@Model.Price.ToString("c")</smal
l>
            </span>
        </h4>
    </div>
    <form id="@Model.ProductID" asp-action="AddToCart"
            asp-controller="Cart" method="post">
        <input type="hidden" asp-for="ProductID" />
        <input type="hidden" name="returnUrl"
               value="@ViewContext.HttpContext.Request.
PathAndQuery()" />
        <span class="card-text p-1">
            @Model.Description
            <button type="submit"
                class="btn btn-success btn-sm pull-righ
t" style="float:right">
                Add To Cart
            </button>
        </span>
    </form>
</div>
```

这里添加了一个form元素，其中包含隐藏的input元素，用来指定视图模型中的ProductID值，以及更新购物车后浏览器应返回的URL。form元素和其中一个input元素是使用内置标签助手配置的，这是一种很有用的方式，可以生成包含模型值的表单，并在应用程序中定位控制器和操作，如第24章

所述。另一个input元素使用这里创建的扩展方法来设置返回的URL。这里还添加了一个button元素，用来将表单提交给应用程序。

注　意

这里已将form元素的method属性设置为post，这指示浏览器使用HTTP POST请求提交表单数据。可以更改此设置，以便表单使用GET方法，但应该慎重考虑。HTTP规范要求GET请求必须是幂等的，这意味着它们不能导致更改，但将商品添加到购物车绝对是一种更改。第16章将介绍有关这个主题的更多内容，包括解释如果忽略幂等GET请求的需求，可能会发生什么情况。

9.2.3　启用会话

这里将使用会话状态存储用户购物车的详细信息，会话状态是存储在服务器上并与用户发出的一系列请求相关联的数据。ASP.NET提供了多种不同方式来存储会话状态，包括将其存储在内存中，这也是这里将要使用的方式。这种方式的优点是简单，但意味着当应用程序停止或重新启动时会话数据会丢失。启用会话需要在Startup类中添加服务和中间件，如代码清单9-17所示。

代码清单9-17　在SportsStore文件夹下的Startup.cs文件中启用会话

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));
            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvc(routes => {
```

```
                // ...routing configuration omitted for
brevity...

            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

可调用AddMemoryCache方法来设置内存数据存储。AddSession方法用来注册用于访问会话数据的服务，UseSession方法允许会话系统在从客户端到达时自动将请求与会话相关联。

9.2.4　实现Cart控制器

这里需要一个控制器来处理Add to Cart按钮。在Controllers文件夹中添加一个名为CartController.cs的类文件，并用它定义代码清单9-18所示的类。

代码清单9-18　Controllers文件夹下的CartController.cs文件的内容

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo)
{
            repository = repo;
        }

        public RedirectToActionResult AddToCart(int pro
ductId, string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId);

            if (product != null) {
                Cart cart = GetCart();
                cart.AddItem(product, 1);
                SaveCart(cart);
            }
            return RedirectToAction("Index", new { retu
rnUrl });
        }

        public RedirectToActionResult RemoveFromCart(in
t productId,
        string returnUrl) {
            Product product = repository.Products
```

```
                .FirstOrDefault(p => p.ProductID == pro
ductId);

            if (product != null) {
                Cart cart = GetCart();
                cart.RemoveLine(product);
                SaveCart(cart);
            }
            return RedirectToAction("Index", new { retu
rnUrl });
        }

        private Cart GetCart() {
            Cart cart = HttpContext.Session.GetJson<Car
t>("Cart") ?? new Cart();
            return cart;
        }

        private void SaveCart(Cart cart) {
            HttpContext.Session.SetJson("Cart", cart);
        }
    }
}
```

关于这个控制器有几点需要注意。首先，使用ASP.NET会话状态功能来存储和检索Cart对象，这是GetCart方法的目的。前面在9.2.3节中注册的中间件使用Cookie或URL重写将来自一个用户的多个请求关联在一起以形成单个浏览会话。相关的功能是

会话状态，用于将数据与会话相关联。这是Cart类的理想选择：希望每个用户都有自己的购物车，并且希望购物车在请求之间保持一致。会话过期时会删除与会话关联的数据（通常是因为用户在一段时间内没有发出任何请求），这意味着不需要管理Cart对象的存储或生命周期。

对于AddToCart和RemoveFromCart操作方法，可以使用与ProductSummary.cshtml视图文件中创建的HTML表单中的input元素匹配的参数名称。这允许MVC将表单传入的POST变量与这些参数相关联，进而意味着不需要自己处理表单。这称为模型绑定，模型绑定简化控制器类的强大工具，正如第26章解释的那样。

定义会话状态扩展方法

ASP.NET Core中的会话状态功能仅存储int、

string和byte[]值。由于需要存储Cart对象，因此需要为ISession接口定义扩展方法，ISession接口提供对会话状态数据的访问，以将Cart对象序列化为JSON并将其转换回来。在Infrastructure文件夹中添加名为SessionExtensions.cs的类文件，并定义代码清单9-19所示的扩展方法。

代码清单9-19　Infrastructure文件夹下的SessionExtensions.cs文件的内容

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace SportsStore.Infrastructure {

    public static class SessionExtensions {

        public static void SetJson(this ISession sessio
n, string key, object value) {
            session.SetString(key, JsonConvert.Serializ
eObject(value));
        }

        public static T GetJson<T>(this ISession sessio
n, string key) {
            var sessionData = session.GetString(key);
            return sessionData == null
```

```
            ? default(T) : JsonConvert.DeserializeObjec
t<T>(sessionData);
        }
    }
}
```

这些方法依赖于Json.Net包并将对象序列化为JavaScript Object Notation格式，你将在第20章中再次遇到这种格式。Json.Net包不必添加到项目中，因为MVC已经默认集成了它，用来提供JSON助手功能，如第21章所述（有关使用Json.Net包的信息，请参阅newtonsoft网站）。

扩展方法可以轻松存储和检索Cart对象。要在控制器中将Cart添加到会话状态，可以这样做：

```
...
HttpContext.Session.SetJson("Cart", cart);
...
```

HttpContext属性由Controller基类提供，控制器通常从Controller基类派生，并返回一个HttpContext

对象，该对象提供有关已接收的请求和正在准备的响应的上下文数据。HttpContext.Session属性会返回一个实现了ISession接口的对象，ISession接口定义了SetJson方法的类型，SetJson方法接收一些参数，指定了一个键和要添加到会话状态的对象。扩展方法会序列化对象，并使用ISession接口提供的基础功能将其添加到会话状态中。

再次检索购物车，使用另一种扩展方法，但指定相同的键，如下所示：

```
...
Cart cart = HttpContext.Session.GetJson<Cart>("Cart");
...
```

类型参数用于指定要检索的类型，用于反序列化过程。

9.2.5 显示购物车的内容

关于Cart控制器的最后一点是，AddToCart和RemoveFromCart方法都调用RedirectToAction方法。这具有向客户端浏览器发送HTTP重定向指令的效果，要求浏览器请求新的URL。在这种情况下，我们已经要求浏览器请求一个URL，该URL将调用Cart控制器的Index操作方法。

实现Index操作方法并用它来显示购物车的内容。如果回头参考图9-7，你将看到这是用户单击Add to Cart按钮时的工作流程。

需要将两条信息传递给显示购物车内容的视图，分别是Cart对象以及用户单击Continue Shopping按钮时显示的URL。在SportsStore项目的Models/ViewModels文件夹中创建名为CartIndexViewModel.cs的类文件，并用它定义代码清单9-20所示的类。

代码清单9-20　Models/ViewModels文件夹下的CartIndexViewModel.cs文件的内容

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    public class CartIndexViewModel {
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

现在有了视图模型，可以在Cart控制器类中实现Index操作方法了，如代码清单9-21所示。

代码清单9-21　在Controllers文件夹下的CartController.cs文件中实现Index操作方法

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class CartController : Controller {
```

```
        private IProductRepository repository;

        public CartController(IProductRepository repo)
{
            repository = repo;
        }

        public ViewResult Index(string returnUrl) {
            return View(new CartIndexViewModel {
                Cart = GetCart(),
                ReturnUrl = returnUrl
            });
        }

        // ...other methods omitted for brevity...
    }
}
```

Index操作方法从会话状态检索Cart对象，并使用它创建CartIndexViewModel对象，然后将其作为视图模型传递给View方法。

显示购物车内容的最后一步是创建Index操作方法将要呈现的视图。创建Views/Cart文件夹，并在其中添加一个名为Index.cshtml的Razor视图文件，其中的内容如代码清单9-22所示。

代码清单9-22　Views/Cart文件夹下的Index.cshtml文件的内容

```
@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td class="text-center">@line.Quantity<
/td>
                <td class="text-left">@line.Product.Nam
e</td>
                <td class="text-right">@line.Product.Pr
ice.ToString("c")</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Pri
ce).ToString("c"))
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</
td>
```

```
            <td class="text-right">
                @Model.Cart.ComputeTotalValue().ToStrin
g("c")
            </td>
        </tr>
    </tfoot>
</table>

<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">
Continue Shopping</a>
</div>
```

这个视图枚举了购物车中的行，并将其中的每行以及每行的总价和购物车的总价添加到HTML表格中。为了让表格和文本对齐，可以为这些元素分配与Bootstrap样式对应的class样式。

购物车的基本功能已完成。首先，列出的商品旁边有一个按钮，用来将它们添加到购物车中，如图9-8所示。

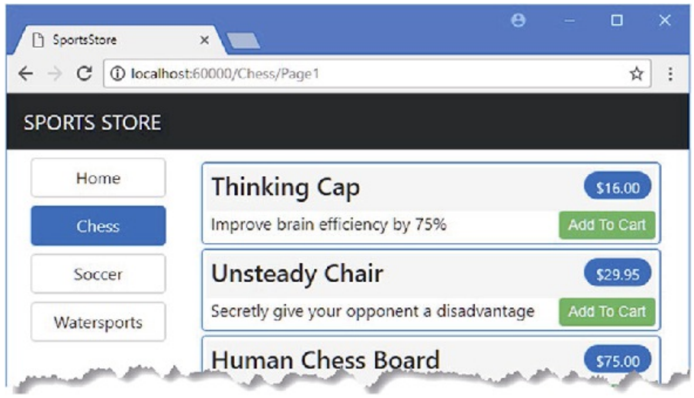


图9-8　Add to Cart按钮

其次，当用户单击Add to Cart按钮时，相应的商品将被添加到购物车中，并显示购物车的摘要，如图9-9所示。单击Continue Shopping按钮可让用户返回之前的商品页面。

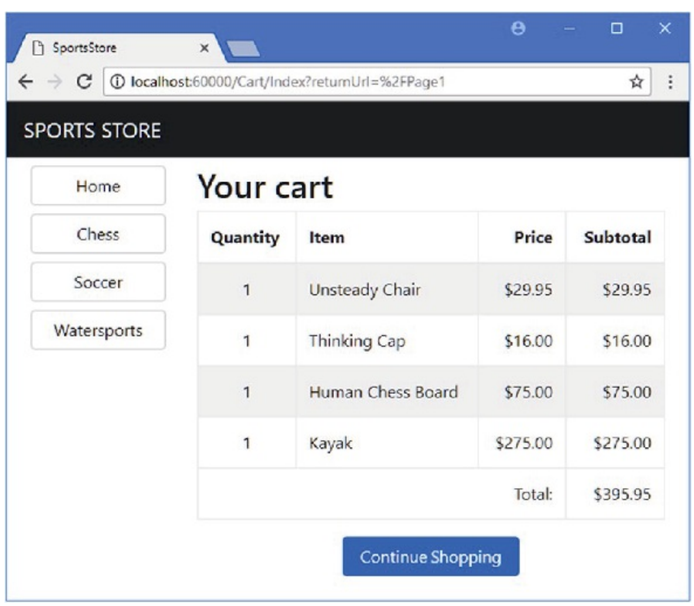


图9-9　显示购物车的内容

9.3　小结

本章介绍了如何实现SportsStore应用程序面向客户的部分。我们为用户提供了可以按类别导航的方法，并将基础构建模块准备就绪，以便将商品添加到购物车。此外，还有更多工作要做，我们将在下一章继续开发该应用程序。

第10章　完成购物车

本章将继续介绍如何构建SportsStore示例应用程序。在上一章中，我们添加了购物车的基本功能，现在将改进并完成这些功能。

10.1　使用服务优化购物车模型

上一章定义了Cart模型类，并展示了如何使用会话功能来存储它，从而允许用户保存一组已购买的商品。Cart类的持久化是由Cart控制器管理的，Cart控制器明确地定义了获取和存储Cart对象的方法。

这种方式的问题是，为了获取并存储Cart对象，必须在所有使用它们的组件中复制这些代码。在本节中，将使用ASP.NET Core的服务功能来简化

Cart对象的管理方式，释放诸如Cart控制器的各个组件，从而无须直接处理细节。

服务的最常用方式是隐藏接口从依赖于它们的组件进行实例化的细节。你已经看到了一个例子，就是为IProductRepository接口创建一个服务，这使你能够无缝地将虚拟存储库类用Entity Framework Core存储库替换掉。但服务也可用于解决许多其他问题，并且可以用于组成和重构应用程序，即使正在使用具体的类，如Cart类。

10.1.1　创建支持存储感知的Cart类

整理Cart类的使用方式的第一步是创建一个子类，该子类能够知道如何使用会话状态来存储自己。添加一个名为SessionCart.cs的类文件到Models文件夹中，并用它定义代码清单10-1所示的类。

代码清单10-1　Models文件夹下的SessionCart.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using SportsStore.Infrastructure;

namespace SportsStore.Models {

    public class SessionCart: Cart {
        public static Cart GetCart(IServiceProvider ser
vices) {
            ISession session = services.GetRequiredServ
ice<IHttpContextAccessor>()?
                .HttpContext.Session;
            SessionCart cart = session?.GetJson<Session
Cart>("Cart")
                ?? new SessionCart();
            cart.Session = session;
            return cart;
        }

        [JsonIgnore]
        public ISession Session { get; set; }

        public override void AddItem(Product product, i
nt quantity) {
            base.AddItem(product, quantity);
            Session.SetJson("Cart", this);
        }

        public override void RemoveLine(Product product
) {
            base.RemoveLine(product);
            Session.SetJson("Cart", this);
        }
```

```
        public override void Clear() {
            base.Clear();
            Session.Remove("Cart");
        }
    }
}
```

SessionCart类是Cart类的子类，并且重写了AddItem、RemoveLine和Clear方法，从而调用基本实现，然后使用第9章定义的ISession接口上的扩展方法将更新后的状态存储在会话中。静态方法GetCart是一个工厂，用于创建SessionCart对象并为它们提供一个ISession对象，以便它们可以存储自身。掌握ISession对象有点复杂。必须获得IHttpContextAccessor服务的一个实例，该服务提供对HttpContext对象的访问，而HttpContext对象又提供了ISession。这种间接方法是必需的，因为会话不是作为常规服务提供的。

10.1.2　注册服务

下一步是为Cart类创建一个服务。目标是使用SessionCart对象来满足Cart对象的请求，这些对象将无缝地存储自身。代码清单10-2展示了如何创建服务。

代码清单10-2　在SportsStore文件夹下的Startup.cs文件中创建购物车服务

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration["Data:SportStoreProducts:ConnectionString"]));
    services.AddTransient<IProductRepository, EFProductRepository>();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddMvc();
    services.AddMemoryCache();
    services.AddSession();
}
...
```

AddScoped方法指定应使用相同的对象来满足Cart实例的相关请求。请求之间的关系是可以配置的，但默认情况下，这意味着处理相同HTTP请求的组件所需的任何Cart都将接收到相同的对象。

这里没有像为存储库那样为AddScoped方法提供类型映射的AddScoped方法，而是指定了一个Lambda表达式，这个Lambda表达式将被调用以满足Cart请求。这个Lambda表达式接收已注册的服务集合，并将集合传递给SessionCart类的GetCart方法。结果是，对Cart服务的请求将通过创建SessionCart对象来处理，当这些对象被修改时，会将自己序列化为会话数据。

这里还使用AddSingleton方法添加了一个服务，该方法指定应始终使用相同的对象，该服务告诉MVC在需要IHttpContextAccessor接口的实现时使用HttpContextAccessor类。该服务是必需的，因此

在这里可以访问代码清单10-1中的SessionCart类中的当前会话。

10.1.3　简化Cart控制器

创建这种服务的好处是能够简化使用Cart对象的控制器。通过代码清单10-3，重新设计CartController类以利用新服务。

代码清单10-3　在Controllers文件夹下的CartController.cs文件中使用购物车服务

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private Cart cart;

        public CartController(IProductRepository repo,
Cart cartService) {
            repository = repo;
            cart = cartService;
```

```
        }

        public ViewResult Index(string returnUrl) {
            return View(new CartIndexViewModel {
                Cart = cart,
                ReturnUrl = returnUrl
            });
        }
        public RedirectToActionResult AddToCart(int pro
ductId, string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId);
            if (product != null) {
                cart.AddItem(product, 1);
            }
            return RedirectToAction("Index", new { retu
rnUrl });
        }

        public RedirectToActionResult RemoveFromCart(in
t productId,
                string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId);

            if (product != null) {
                cart.RemoveLine(product);
            }
            return RedirectToAction("Index", new { retu
rnUrl });
        }
    }
}
```

CartController类通过声明构造函数的参数来指示这一个Cart对象，这样就可以删除从会话读取和写入数据的方法以及写入更新所需的步骤。其结果是一个更简单的控制器，并且仍然专注于它在应用程序中担负的角色，而不必担心如何创建或持久化Cart对象。此外，由于服务在整个应用程序中都可以使用，因此任何组件都可以使用相同的技术来处理用户的购物车。

10.2　完成购物车功能

既然上一节已经介绍了购物车服务，现在是时候通过添加两个新功能来完成购物车功能了。第一个功能将允许客户从购物车中删除商品。第二个功能将在页面顶部显示购物车的摘要。

10.2.1　从购物车中删除商品

由于已经在控制器中定义并测试了RemoveFromCart操作方法，因此只需要在视图中公开这个操作方法即可让客户从购物车中删除商品，可通过在购物车摘要的每一行中添加Remove按钮来完成此操作。代码清单10-4显示了对Views/Cart/Index.cshtml文件所做的更改。

代码清单10-4　将Remove按钮引入Views/Cart文件夹下的Index.cshtml文件

```
@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td class="text-center">@line.Quantity<
/td>
```

```
                <td class="text-left">@line.Product.Nam
e</td>
                <td class="text-right">@line.Product.Pr
ice.ToString("c")</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Pri
ce).ToString("c"))
                </td>
                <td>
                    <form asp-action="RemoveFromCart" m
ethod="post">
                        <input type="hidden" name="Prod
uctID"
                               value="@line.Product.Pro
ductID" />
                        <input type="hidden" name="retu
rnUrl"
                               value="@Model.ReturnUrl"
 />
                        <button type="submit" class="bt
n btn-sm btn-danger">
                            Remove
                        </button>
                    </form>
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</
td>
            <td class="text-right">
                @Model.Cart.ComputeTotalValue().ToStrin
g("c")
            </td>
```

```
        </tr>
    </tfoot>
</table>

<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">
Continue shopping</a>
</div>
```

以上代码在表格的每一行中添加了一个新列，其中包含一个带有隐藏的input元素的form表单，用于指定要删除的商品和要返回的URL，以及用于提交表单的按钮。

通过运行应用程序并将商品添加到购物车中，可以查看Remove按钮如何工作。请记住，购物车已经包含删除商品的功能，可以通过单击其中一个新的按钮来测试，如图10-1所示。

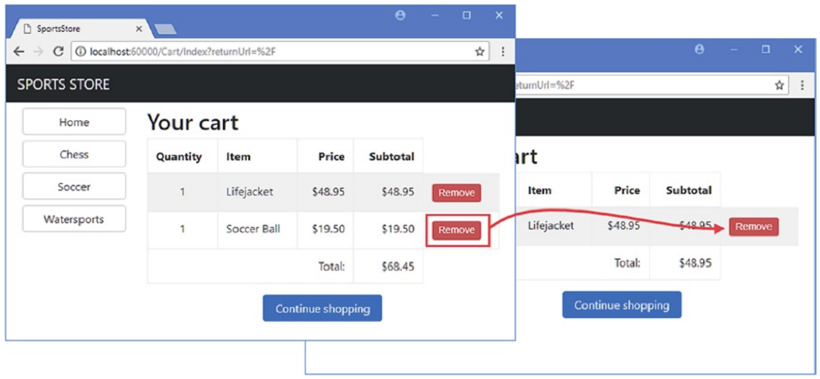


图10-1　从购物车中删除商品

10.2.2　添加购物车摘要小部件

虽然已经有了功能正常的购物车，但也有一个问题：客户只能查看购物车摘要页面才能了解购物车中的商品，并且他们只能通过向购物车添加新商品来查看购物车摘要页面。

为了解决这个问题，可添加一个小部件，简要显示购物车的内容，并且单击该小部件就可以在整个应用程序中显示购物车的内容。我们以与添加导航小部件相同的方式执行此操作，作为视图组件，其输出可以包含在Razor共享布局中。

1．添加**Font Awesome**包

作为购物车摘要的一部分，需要显示一个允许用户结账的按钮。这里想使用购物车图标，而不是在按钮上显示单词Checkout。Font Awesome包提供了一组优秀的开源图标，可以作为字体集成到应用程序中，字体中的每个字符都是不同的图像。可以在GitHub上了解有关Font Awesome包的更多信息，包括查看其中包含的图标。这里选择SportsStore项目，然后单击Solution Explorer窗格顶部的Show All Items按钮以显示bower.json文件。接下来，将Font Awesome包添加到dependencies部分，如代码清单10-5所示。

代码清单10-5　在SportsStore文件夹下的bower.json文件中添加Font Awesome包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
```

```
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0"
  }
}
```

保存bower.json文件后，Visual Studio使用Bower在www/lib/fontawesome文件夹中下载并安装Font Awesome包。

2．创建视图组件类和视图

下面在Components文件夹中添加名为CartSummaryViewComponent.cs的类文件，并用它定义代码清单10-6所示的视图组件。

代码清单10-6　Components文件夹下的CartSummaryViewComponent.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Components {

    public class CartSummaryViewComponent : ViewCompone
nt {
```

```
        private Cart cart;

        public CartSummaryViewComponent(Cart cartServic
e) {
            cart = cartService;
        }

        public IViewComponentResult Invoke() {
            return View(cart);
        }
    }
}
```

这个视图组件能够利用本章前面创建的服务，将Cart对象作为构造函数的参数接收。结果是一个简单的视图组件类，它将Cart对象传递给View方法，以便生成包含在布局中的HTML片段。为了创建布局，创建Views/Shared/Components/CartSummary文件夹，在其中添加名为Default.cshtml的Razor视图文件，并添加代码清单10-7所示的标记。

代码清单10-7　Views/Shared/Components/CartSummary文件夹下的Default.cshtml文件的内容

```
@model Cart

<div class="">
    @if (Model.Lines.Count() > 0) {
        <small class="navbar-text">
            <b>Your cart:</b>
            @Model.Lines.Sum(x => x.Quantity) item(s)
            @Model.ComputeTotalValue().ToString("c")
        </small>
    }
    <a class="btn btn-sm btn-secondary navbar-btn"
        asp-controller="Cart" asp-action="Index"
        asp-route-returnurl="@ViewContext.HttpContext.R
equest.PathAndQuery()">
         <i class="fa fa-shopping-cart"></i>
    </a>
</div>
```

视图将显示带有Font Awesome购物车图标的按钮，如果购物车中有商品，则会提供一个页面，详细说明商品的数量及总价。现在有了视图组件和视图，接下来可以修改共享布局，以将购物车摘要包含在应用程序控制器生成的响应中，如代码清单10-8所示。

代码清单10-8　在Views/Shared文件夹下的_Layout.cshtml文件中添加购物车摘要

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet"
          asp-href-include="/lib/bootstrap/dist/**/*.mi
n.css"
          asp-href-exclude="**/*-reboot*,**/*-grid*" />
    <link rel="stylesheet" asp-href-include="/lib/fonta
wesome/css/*.css" />
    <title>SportsStore</title>
</head>
<body>
    <div class="navbar navbar-inverse bg-inverse" role=
"navigation">
        <div class="row">
            <a class="col navbar-brand" href="#">SPORTS
 STORE</a>
            <div class="col-4 text-right">
                @await Component.InvokeAsync("CartSumma
ry")
            </div>
        </div>
    </div>
    <div class="row m-1 p-1">
        <div id="categories" class="col-3">
            @await Component.InvokeAsync("NavigationMen
u")
        </div>
        <div class="col-9">
            @RenderBody()
        </div>
    </div>
```

```
</body>
</html>
```

可以通过启动应用程序来查看购物车摘要。当购物车为空时，仅显示Checkout按钮。如果将商品添加到了购物车中，则会显示商品的数量及总价，如图10-2所示。通过以上修改，客户可以了解购物车中的内容，并可以方便地结账。

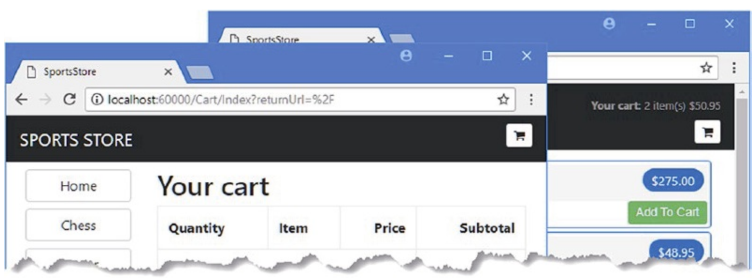


图10-2　显示购物车摘要

10.3　提交订单

现在已经完成了SportsStore应用程序最终面向客户的功能——检查并完成订单。本节将介绍如何扩展域模型，支持获取用户的装运信息，并添加应

用程序支持以处理它们。

10.3.1　创建模型类

在Models文件夹中添加一个名为Order.cs的类文件，内容如代码清单10-9所示。这是用于表示客户装运信息的类。

代码清单10-9　Models文件夹下的Order.cs文件的内容

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Order {

        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set;
}

        [Required(ErrorMessage = "Please enter a name")
]
        public string Name { get; set; }
```

```
        [Required(ErrorMessage = "Please enter the firs
t address line")]
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }
        [Required(ErrorMessage = "Please enter a city n
ame")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state
name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a countr
y name")]
        public string Country { get; set; }

        public bool GiftWrap { get; set; }
    }
}
```

使用System.ComponentModel.DataAnnotations命名空间中的验证属性，就像在第2章中所做的那样。第27章将进一步描述验证。

这里还使用了BindNever特性，该特性会阻止用户在HTTP请求中为这些属性提供值。这是在第

26章将要描述的模型绑定系统的特性之一——阻止MVC使用HTTP请求中的值来填充敏感或重要的模型属性。

10.3.2　添加结账流程

为了使用户能够输入装运信息并提交订单，首先需要在购物车的摘要视图中添加Checkout按钮。代码清单10-10显示了对Views/Cart/Index.cshtml文件所做的更改。

代码清单10-10　将Checkout按钮添加到Views/Cart文件夹下的Index.cshtml文件中

```
...
<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">
Continue Shopping</a>
    <a class="btn btn-primary" asp-action="Checkout" as
p-controller="Order">
        Checkout
    </a>
</div>
...
```

以上更改会生成一个链接，将其样式化为按钮，并在单击时调用Order控制器的Checkout操作方法。可以在图10-3中看到Checkout按钮的显示方式。

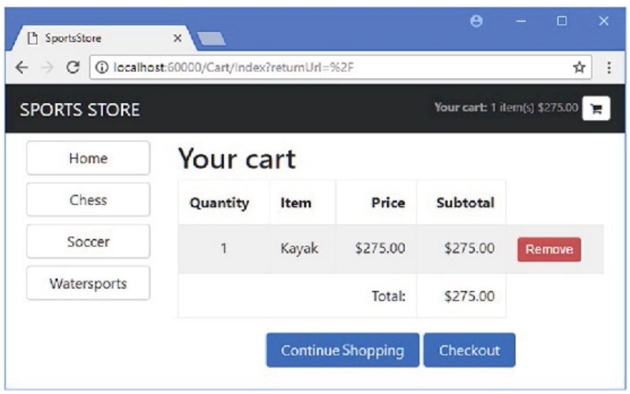


图10-3　Checkout按钮

现在需要定义Order控制器。在Controllers文件夹中添加一个名为OrderController.cs的类文件，并用它定义代码清单10-11所示的类。

代码清单10-11　Controllers文件夹下的OrderController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
```

```
namespace SportsStore.Controllers {

    public class OrderController : Controller {

        public ViewResult Checkout() => View(new Order(
));
    }
}
```

Checkout方法返回默认视图，并传递一个新的ShippingDetails对象作为视图模型。为了创建视图，创建Views/Order文件夹并添加一个名为Checkout.cshtml的Razor视图文件，其中包含代码清单10-12所示的内容。

代码清单10-12　Views/Order文件夹下的Checkout.cshtml文件的内容

```
@model Order

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods
 right away!</p>

<form asp-action="Checkout" method="post">
    <h3>Ship to</h3>
    <div class="form-group">
```

```
        <label>Name:</label><input asp-for="Name" class
="form-control" />
    </div>
    <h3>Address</h3>
    <div class="form-group">
        <label>Line 1:</label><input asp-for="Line1" cl
ass="form-control" />
    </div>
    <div class="form-group">
        <label>Line 2:</label><input asp-for="Line2" cl
ass="form-control" />
    </div>
    <div class="form-group">
        <label>Line 3:</label><input asp-for="Line3" cl
ass="form-control" />
    </div>
    <div class="form-group">
        <label>City:</label><input asp-for="City" class
="form-control" />
    </div>
    <div class="form-group">
        <label>State:</label><input asp-for="State" cla
ss="form-control" />
    </div>
    <div class="form-group">
        <label>Zip:</label><input asp-for="Zip" class="
form-control" />
    </div>
    <div class="form-group">
        <label>Country:</label><input asp-for="Country"
 class="form-control" />
    </div>
    <h3>Options</h3>
    <div class="checkbox">
        <label>
            <input asp-for="GiftWrap" /> Gift wrap thes
```

```
e items
        </label>
    </div>
    <div class="text-center">
        <input class="btn btn-primary" type="submit" va
lue="Complete Order" />
    </div>
</form>
```

对于模型中的每个属性，创建Bootstrap样式的label元素和input元素来捕获用户输入。input元素的asp-for属性由内置的标签助手处理，标签助手将根据指定的模型属性生成type、id、name和value属性，如第24章所述。

你可以通过启动应用程序，单击页面顶部的购物车图标，然后单击Checkout按钮来查看操作方法和视图的效果，如图10-4所示。你也可以通过请求/Cart/Checkout的URL来查看这个页面。

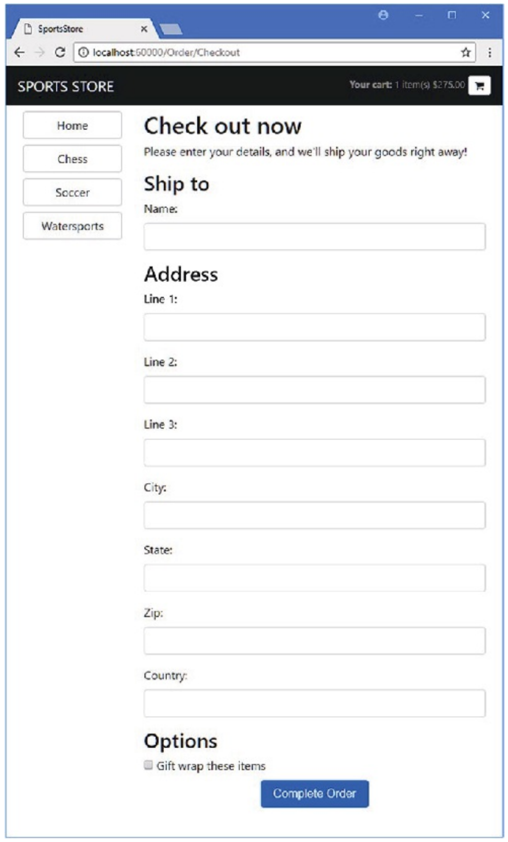


图10-4　装运信息表单

10.3.3　实现订单处理

可通过将订单写入数据库来处理订单。当然，大多数电子商务网站不会简单地如此实现，而且这里没有提供处理信用卡或其他付款方式的功能。但是因为这里想把重点放在MVC上，所以一个简单的数据库条目就可以了。

1．扩展数据库

如果在第8章中创建的基本管道程序就位了，就可以将新的模型添加到数据库中。首先，向数据库上下文类添加一个新的属性，如代码清单10-13所示。

代码清单10-13　在Models文件夹下的ApplicationDbContext.cs文件中添加属性

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions<Ap
plicationDbContext> options)
            : base(options) { }

        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

上述更改能够让Entity Framework Core创建数据库迁移，以将Order对象存储在数据库中。要创建数据库迁移，请打开新的命令提示符或PowerShell窗口，导航到SportsStore项目文件夹（其中包含Startup.cs文件），然后运行以下命令：

```
dotnet ef migrations add Orders
```

上述命令告诉Entity Framework Core获取应用程序数据模型的新快照，并对比与以前数据库版本的不同之处，生成名为Orders的数据库迁移。应用程序在启动时将自动应用新的数据库迁移，因为SeedData会调用Entity Framework Core提供的Migrate方法。

重置数据库

当频繁更改模型时，数据库迁移和数据库架构不同步时会出现问题。最简单的方法是删除数据库并重新开始。但是这仅适用于开发期间，因为你会丢失已存储的任何数据。

要删除数据库，请在SportsStore项目文件夹中运行以下命令：

```
dotnet ef database drop --force
```

删除数据库后，从SportsStore项目文件夹运行以下命令以重新创建数据库，并通过运行以下命令应用创建的数据库迁移：

```
dotnet ef database update
```

这将重置数据库，以准确反映模型的更改，这样你就可以继续开发应用程序了。

2．创建订单存储库

下面沿用产品存储库的模式，以提供对Order对象的访问。在Models文件夹中添加一个名为IOrderRepository.cs的类文件，并用它定义代码清单10-14所示的接口。

代码清单10-14　Models文件夹下的IOrderRepository.cs文件的内容

```
using System.Linq;

namespace SportsStore.Models {

    public interface IOrderRepository {

        IQueryable<Order> Orders { get; }
        void SaveOrder(Order order);
    }
}
```

为了实现订单存储库接口，将一个名为EFOrderRepository.cs的类文件添加到Models文件夹中，并定义代码清单10-15所示的类。

代码清单10-15　Models文件夹下的EFOrderRepository.cs文件的内容

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace SportsStore.Models {

    public class EFOrderRepository : IOrderRepository {
        private ApplicationDbContext context;

        public EFOrderRepository(ApplicationDbContext c
tx) {
            context = ctx;
        }
        public IQueryable<Order> Orders => context.Orde
rs
                            .Include(o => o.Lines)
                            .ThenInclude(l => l.Product
);

        public void SaveOrder(Order order) {
            context.AttachRange(order.Lines.Select(l =>
 l.Product));
            if (order.OrderID == 0) {
                context.Orders.Add(order);
            }
            context.SaveChanges();
        }
    }
}
```

这个类使用Entity Framework Core实现了IOrderRepository，从而能够检索已存储的Order对象集，并允许创建或更改订单。

了解订单存储库

为代码清单10-15中的订单实现存储库需要做一些额外的工作。Entity Framework Core需要指令来加载相关数据（如果跨越多个表的话）。在代码中，使用Include和ThenInclude方法指定从数据库中读取Order对象时，还应该一起加载与Lines属性关联的集合以及与每个集合对象关联的Product对象。

```
...
public IQueryable<Order> Orders => context.Orders
    .Include(o => o.Lines)
    .ThenInclude(l => l.Product);
...
```

这将确保收到所需的所有数据对象，而无须执行查询和重新组装数据。

当在数据库中存储Order对象时，还需要执行一个额外的步骤。当从会话存储中反序列化用户的购物车数据时，JSON包会创建Entity Framework Core不知道的新对象，然后尝试将所有对象写入数据库。对于Product对象，这意味着Entity Framework Core尝试写入已存储的对象，这会导致错误。为了避免这个问题，可通知Entity Framework Core对象已存在，除非它们被修改，否则不应该存储在数据库中，如下所示：

```
...
context.AttachRange(order.Lines.Select(l => l.Product))
;
...
```

这可确保Entity Framework Core不会尝试写入与Order对象关联的反序列化后的Product对象。

在代码清单10-16中，我们已在Startup类的ConfigureServices方法中将订单存储库注册为服务。

代码清单10-16　在SportsStore文件夹下的Startup.cs文件中注册Order存储库服务

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration["Data:SportStoreProducts:ConnectionString"]));
    services.AddTransient<IProductRepository, EFProductRepository>();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddTransient<IOrderRepository, EFOrderRepository>();
    services.AddMvc();
    services.AddMemoryCache();
    services.AddSession();
}
```

```
...
```

10.3.4　完成Order控制器

为了完成OrderController类，还需要修改构造函数，以便接收处理订单所需的服务，并且需要添加一个新的操作方法。当用户单击Complete Order按钮时，这个操作方法将处理HTTP表单的POST请求。代码清单10-17显示了这两处更改。

代码清单10-17　在Controllers文件夹下的OrderController.cs文件中完成Order控制器

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;

        public OrderController(IOrderRepository repoSer
vice, Cart cartService) {
            repository = repoService;
```

```
            cart = cartService;
        }

        public ViewResult Checkout() => View(new Order(
));

        [HttpPost]
        public IActionResult Checkout(Order order) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, yo
ur cart is empty!");
            }
            if (ModelState.IsValid) {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Complete
d));
            } else {
                return View(order);
            }
        }

        public ViewResult Completed() {
            cart.Clear();
            return View();
        }
    }
}
```

Checkout操作方法使用HttpPost特性进行修饰，这意味着仅为POST请求调用这个操作方法。在本例中，也就是当用户提交表单时。再次使用模

型绑定系统以接收Order对象，然后处理Cart中的数据并存储在存储库中。

MVC使用数据注解属性检查应用于Order类的验证约束，并且任何验证相关的问题都会通过ModelState属性传递给操作方法。可以通过检查ModelState.IsValid属性来查看是否存在问题。如果购物车中没有商品，就调用ModelState.AddModelError方法来注册错误消息。稍后会解释如何显示此类错误，第27章和第28章将介绍更多关于模型绑定和验证的内容。

单元测试——订单处理

为了对OrderController类执行单元测试，需要测试Checkout方法的POST请求的行为。尽管Checkout方法看起来很简单，但使用MVC模型绑定

意味着对测试来说还有很多事情要做。

我们想仅在购物车中有商品并且客户提供有效的装运信息时处理订单，而在其他情况下应向客户显示错误。在SportsStore.Test项目中，在名为OrderControllerTests.cs的类文件中定义第一个测试方法。

```
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class OrderControllerTests {

        [Fact]
        public void Cannot_Checkout_Empty_Cart() {
            // Arrange - create a mock repository
            Mock<IOrderRepository> mock = new Mock<IOrd
erRepository>();
            // Arrange - create an empty cart
            Cart cart = new Cart();
            // Arrange - create the order
            Order order = new Order();
            // Arrange - create an instance of the cont
```

```
roller
            OrderController target = new OrderControlle
r(mock.Object, cart);

            // Act
            ViewResult result = target.Checkout(order)
as ViewResult;

            // Assert - check that the order hasn't bee
n stored
            mock.Verify(m => m.SaveOrder(It.IsAny<Order
>()), Times.Never);
            // Assert - check that the method is return
ing the default view
            Assert.True(string.IsNullOrEmpty(result.Vie
wName));
            // Assert - check that I am passing an inva
lid model to the view
            Assert.False(result.ViewData.ModelState.IsV
alid);
        }
    }
}
```

以上测试可确保在购物车为空时不能结账。可通过确保不调用模拟IOrderRepository实现的SaveOrder方法来检查这一点，该方法返回的是默认视图（它将重新显示客户输入的数据并给他们提供纠正的机会），并且传递给视图的模型状态已被标

记为无效。这可能看起来像是万无一失的断言，但需要确认3个部分以确保得到正确的结果。下一个测试方法的工作方式大致相同，但在视图模型中会注入错误，以模拟模型绑定器报告的问题（当客户输入无效的装运数据时会出现）：

```
...
[Fact]
public void Cannot_Checkout_Invalid_ShippingDetails() {

    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderReposi
tory>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.O
bject, cart);
    // Arrange - add an error to the model
    target.ModelState.AddModelError("error", "error");

    // Act - try to checkout
    ViewResult result = target.Checkout(new Order()) as
 ViewResult;

    // Assert - check that the order hasn't been passed
 stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Ti
mes.Never);
```

```
    // Assert - check that the method is returning the
default view
    Assert.True(string.IsNullOrEmpty(result.ViewName));
    // Assert - check that I am passing an invalid mode
l to the view
    Assert.False(result.ViewData.ModelState.IsValid);
}
...
```

除确保空的购物车或无效的装运信息能阻止继续处理订单之外，还需要确保在适当的时候处理订单。测试如下：

```
...
[Fact]
public void Can_Checkout_And_Submit_Order() {
    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderReposi
tory>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.O
bject, cart);

    // Act - try to checkout
    RedirectToActionResult result =
        target.Checkout(new Order()) as RedirectToActio
nResult;
```

```
    // Assert - check that the order has been stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Ti
mes.Once);
    // Assert - check that the method is redirecting to
 the Completed action
    Assert.Equal("Completed", result.ActionName);
}
...
```

不需要测试那些可以识别的有效的装运细节。这是由模型绑定器自动处理的，可通过在Order类的属性上应用属性注释来实现。

10.3.5　显示验证错误

MVC将使用应用于Order类的验证属性来验证用户数据。但是，还需要进行简单的更改以显示所有问题。这依赖另一个内置的标签助手，它检查用户提供的数据的验证状态，并为已发现的每个问题添加警告消息。代码清单10-18显示了一个HTML元素的附加部分，该HTML元素将由标签助手在

Checkout.cshtml文件中处理。

代码清单10-18　将验证摘要添加到Views/Order文件夹下的Checkout.cshtml文件中

```
@model Order

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods
 right away!</p>

<div asp-validation-summary="All" class="text-danger"><
/div>

<form asp-action="Checkout" method="post">
    <h3>Ship to</h3>
...
```

通过进行上述简单更改，可以向用户报告验证错误。要查看效果，请转到/Order/Checkout URL，并在不选择任何商品或填写任何装运信息的情况下尝试结账，如图10-5所示。生成这些消息的标签助手是模型验证系统的一部分，将在第27章中详细介绍。

提　示

用户提交的数据在验证之前会被发送到服务器，这称为服务器端验证，MVC对此提供良好的支持。服务器端验证的问题是，在将数据发送到服务器并进行处理以生成结果页面之前，不会向用户显示错误——在繁忙的服务器上有时可能需要几秒的时间。因此，服务器端验证通常需要补充使用客户端验证，在将表单数据发送到服务器之前，使用JavaScript检查用户输入的值。第27章将描述客户端验证。

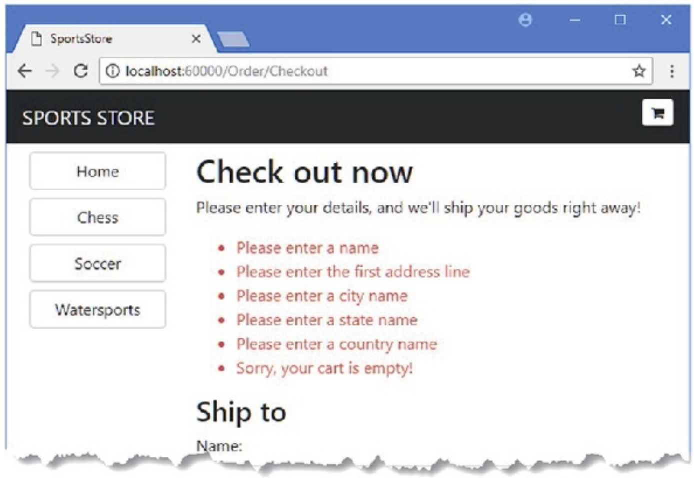


图10-5　显示验证消息

10.3.6　显示摘要页面

为了完成结账流程，需要创建一个视图，在浏览器重定向到Order控制器上的Completed action时显示。在Views/Order文件夹中添加一个名为Completed.cshtml的Razor视图文件，并在其中添加代码清单10-19所示的代码。

代码清单10-19　Views/Order文件夹下的Completed.cshtml文件的内容

```
<h2>Thanks!</h2>
<p>Thanks for placing your order.</p>
```

```
<p>We'll ship your goods as soon as possible.</p>
```

在将这个视图集成到应用程序时，不需要更改任何代码，因为在定义Completed操作方法时已添加必需的语句。现在，客户可以完成从选择商品到结账的整个过程。如果提供有效的装运信息（并且购物车中有商品），在单击Complete Order按钮后就会看到摘要页面，如图10-6所示。

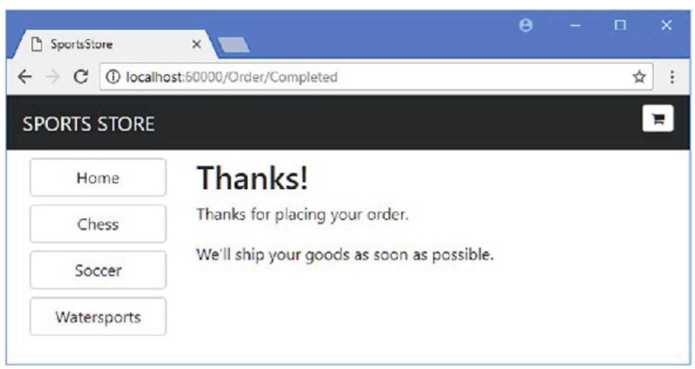


图10-6　已完成的订单摘要页面

10.4　小结

本章介绍了如何实现SportsStore应用程序面向客户的所有主要功能。这里已经有了商品目录，可

以按类别和页面浏览，还提供了简洁的购物车和方便的结账流程。

分层良好的架构意味着可以轻松地更改应用程序的任何部分，而不用担心在其他地方导致问题或不一致。例如，可以更改订单的存储方式，而不会对购物车、商品目录或应用程序的任何其他部分产生影响。在下一章中，将添加管理SportsStore应用程序所需的功能。

第11章　SportsStore的管理

本章将继续介绍如何构建SportsStore应用程序，为站点管理员提供管理订单和商品的功能。

11.1　管理订单

在上一章中，添加了接收客户订单并将其存储在数据库中的功能。在本章中，将创建一个简单的管理工具，用于查看已收到的订单并将其标记为已发货。

11.1.1　增强模型

需要做的第一个更改是增强模型，用来记录已发货的订单。代码清单11-1在Order类中添加了一个新的属性，Order类定义在Models文件夹下的

Order.cs文件中。

代码清单11-1　在Models文件夹下的Order.cs文件中添加一个新的属性

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Order {

        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set;
}

        [BindNever]
        public bool Shipped { get; set; }

        [Required(ErrorMessage = "Please enter a name")
]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the firs
t address line")]
        public string Line1 { get; set; }

        public string Line2 { get; set; }
        public string Line3 { get; set; }
```

```
        [Required(ErrorMessage = "Please enter a city n
ame")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state 
name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a countr
y name")]
        public string Country { get; set; }

        public bool GiftWrap { get; set; }
    }
}
```

这种扩展和调整模型以支持不同功能的迭代方法是MVC开发的典型方式。理想情况下，可以在项目开始时完全定义模型类，并围绕它们构建应用程序，但这只发生在最简单的项目中。实际上，这种迭代开发是必需的，因为对需求的理解一直在发展和演变。

Entity Framework Core迁移使这个过程更容

易，因为不必通过编写SQL命令手动保持数据库架构与模型类同步。要更新数据库以将Shipped属性添加到Order类，请打开新的命令提示符或PowerShell窗口，导航到SportsStore项目文件夹（包含Startup.cs文件的那个文件夹）并运行以下命令：

```
dotnet ef migrations add ShippedOrders
```

当应用程序启动并且SeedData类调用Entity Framework Core提供的Migrate方法时，将自动应用迁移。

11.1.2　添加操作方法和视图

显示和更新数据库中的订单数据相对简单。代码清单11-2在Order控制器中添加了两个新的操作方法。

代码清单11-2　在Controllers文件夹下的OrderController.cs文件中添加操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;
        public OrderController(IOrderRepository repoSer
vice, Cart cartService) {
            repository = repoService;
            cart = cartService;
        }

        public ViewResult List() =>
            View(repository.Orders.Where(o => !o.Shippe
d));

        [HttpPost]
        public IActionResult MarkShipped(int orderID) {
            Order order = repository.Orders
                .FirstOrDefault(o => o.OrderID == order
ID);
            if (order != null) {
                order.Shipped = true;
                repository.SaveOrder(order);
            }
            return RedirectToAction(nameof(List));
        }

        public ViewResult Checkout() => View(new Order(
));

        [HttpPost]
```

```
        public IActionResult Checkout(Order order) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, yo
ur cart is empty!");
            }
            if (ModelState.IsValid) {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Complete
d));
            } else {
                return View(order);
            }
        }

        public ViewResult Completed() {
            cart.Clear();
            return View();
        }
    }
}
```

List操作方法会选择存储库中Shipped属性值为false的所有Order对象，并将它们传递给默认视图。这是用于向管理员显示未发货订单列表的操作方法。

MarkShipped操作方法将接收一个POST请求，

该请求指定了订单的ID，用于从存储库中找到相应的Order对象，以便将其Shipped属性设置为true并保存。

为了显示未发货订单的列表，在Views/Order文件夹中添加一个名为List.cshtml的Razor视图文件，并添加代码清单11-3所示的代码。table元素用于显示一些细节，包括已购买商品的详细信息。

代码清单11-3　Views/Order文件夹下的List.cshtml文件的内容

```
@model IEnumerable<Order>

@{
    ViewBag.Title = "Orders";
    Layout = "_AdminLayout";
}

@if (Model.Count() > 0) {

    <table class="table table-bordered table-striped">
        <tr><th>Name</th><th>Zip</th><th colspan="2">De
tails</th><th></th></tr>
        @foreach (Order o in Model) {
            <tr>
                <td>@o.Name</td><td>@o.Zip</td><th>Prod
```

```
uct</th><th>Quantity</th>
                <td>
                    <form asp-action="MarkShipped" meth
od="post">
                        <input type="hidden" name="orde
rId" value="@o.OrderID" />
                        <button type="submit" class="bt
n btn-sm btn-danger">
                            Ship
                        </button>
                    </form>
                </td>
            </tr>
            @foreach (CartLine line in o.Lines) {
                <tr>
                    <td colspan="2"></td>
                    <td>@line.Product.Name</td><td>@lin
e.Quantity</td>
                    <td></td>
                </tr>
            }
        }
        </table>
} else {
    <div class="text-center">No Unshipped Orders</div>
}
```

每个订单都会显示一个Ship按钮，用于将表单提交给MarkShipped操作方法。使用Layout属性为List视图指定不同的布局，以覆盖_ViewStart.cshtml文件中指定的布局。

要添加布局，可使用MVC View Layout Page模板在Views/Shared文件夹中创建一个名为_AdminLayout. cshtml的文件，并添加代码清单11-4所示的代码。

代码清单11-4　Views/Shared文件夹下的_AdminLayout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></d
iv>
    @RenderBody()
</body>
</html>
```

要查看和管理应用程序中的订单，请启动应用程序，选择一些商品，然后结账。导航

到/Order/List URL，你将看到创建的订单摘要，如图11-1所示。单击Ship按钮，数据库将被更新，待处理的订单列表将为空。

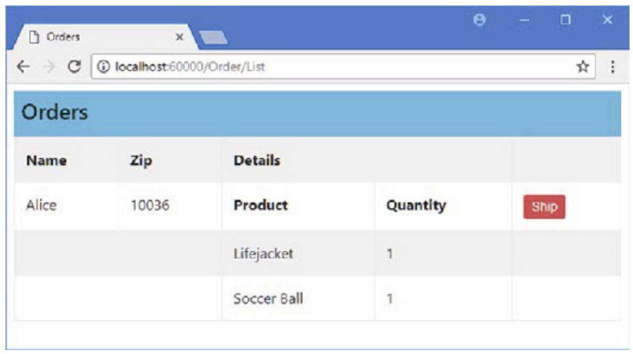


图11-1　管理订单

注　意

目前没有办法阻止客户请求/Order/List URL并管理他们自己的订单。本书第12章将解释如何限制对操作方法的访问。

11.2　添加目录管理

一般来说，在管理复杂的对象集合时，需要向用户呈现两种类型的页面——列表页面和编辑页面，如图11-2所示。

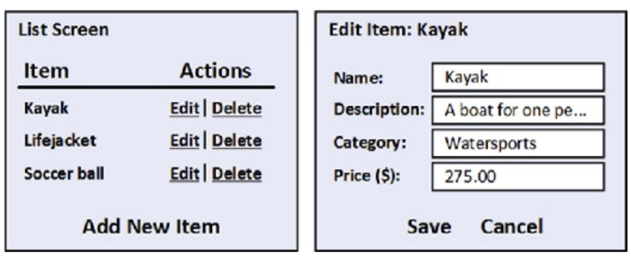


图11-2　商品目录的CRUD UI草图

这些页面允许用户创建、读取、更新和删除集合中的对象。总的来说，这些动作称为CRUD。开发人员经常需要实现CRUD，Visual Studio脚手架包括使用预定义操作方法创建CRUD控制器的场景（第8章解释了如何启用Visual Studio脚手架功能）。但是像所有Visual Studio模板一样，你最好

学习如何直接使用ASP.NET Core MVC的功能。

11.2.1　创建CRUD控制器

这里将首先创建一个单独的控制器来管理商品目录。在Controllers文件夹中添加一个名为AdminController.cs的类文件，并添加代码清单11-5所示的代码。

代码清单11-5　Controllers文件夹下的AdminController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);
```

```
    }
}
```

这个控制器的构造函数声明了对IProductRepository接口的依赖关系，该接口将在创建实例时解析。这个控制器还定义了Index操作方法，这个操作方法会调用View方法以选择操作的默认视图，将数据库中的一组产品作为视图模型传递。

单元测试——**Index**操作方法

Admin控制器的Index操作方法的功能是正确返回存储库中的Product对象。你可以创建模拟的存储库实现，并将测试数据与操作方法返回的数据做比较以进行测试。以下是在SportsStore.UnitTests项目中添加的一个新的名为AdminControllerTests.cs的单

元测试文件：

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class AdminControllerTests {

        [Fact]
        public void Index_Contains_All_Products() {
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IP
roductRepository>();
            mock.Setup(m => m.Products).Returns(new Pro
duct[] {
                new Product {ProductID = 1, Name = "P1"
},
                new Product {ProductID = 2, Name = "P2"
},
                new Product {ProductID = 3, Name = "P3"
},
            }.AsQueryable<Product>());

            // Arrange - create a controller
            AdminController target = new AdminControlle
r(mock.Object);

            // Action
```

```
            Product[] result
                = GetViewModel<IEnumerable<Product>>(ta
rget.Index())?.ToArray();

            // Assert
            Assert.Equal(3, result.Length);
            Assert.Equal("P1", result[0].Name);
            Assert.Equal("P2", result[1].Name);
            Assert.Equal("P3", result[2].Name);
        }

        private T GetViewModel<T>(IActionResult result)
 where T : class {
            return (result as ViewResult)?.ViewData.Mod
el as T;
        }
    }
}
```

在测试中添加GetViewModel方法，以解析操作方法的结果并获取视图模型数据。稍后将添加更多测试。

11.2.2　实现列表视图

下一步是为Admin控制器的Index操作方法添加

视图。创建Views/Admin文件夹并在其中添加一个名为Index.cshtml的Razor文件，其内容如代码清单11-6所示。

代码清单11-6　Views/Admin文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<Product>

@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}

<table class="table table-striped table-bordered table-
sm">
    <tr>
        <th class="text-right">ID</th>
        <th>Name</th>
        <th class="text-right">Price</th>
        <th class="text-center">Actions</th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td class="text-right">@item.ProductID</td>
            <td>@item.Name</td>
            <td class="text-right">@item.Price.ToString
("c")</td>
            <td class="text-center">
                <form asp-action="Delete" method="post"
```

```
>
                    <a asp-action="Edit" class="btn btn
-sm btn-warning"
                       asp-route-productId="@item.Produ
ctID">
                        Edit
                    </a>
                    <input type="hidden" name="ProductI
D" value="@item.ProductID" />
                    <button type="submit" class="btn bt
n-danger btn-sm">
                        Delete
                    </button>
                </form>
            </td>
        </tr>
    }
</table>
<div class="text-center">
    <a asp-action="Create" class="btn btn-primary">Add
Product</a>
</div>
```

列表视图包含一个表格，其中每个商品占据一行，包含商品名称、价格和两个按钮，这两个按钮将能够向Edit和Delete操作发送请求来编辑或删除商品。除表格之外，还有以Create操作为目标的Add Product按钮。后面还将添加Edit、Delete和Create操作，可以通过启动应用程序并请求/Admin/Index

URL来查看商品的显示方式，如图11-3所示。

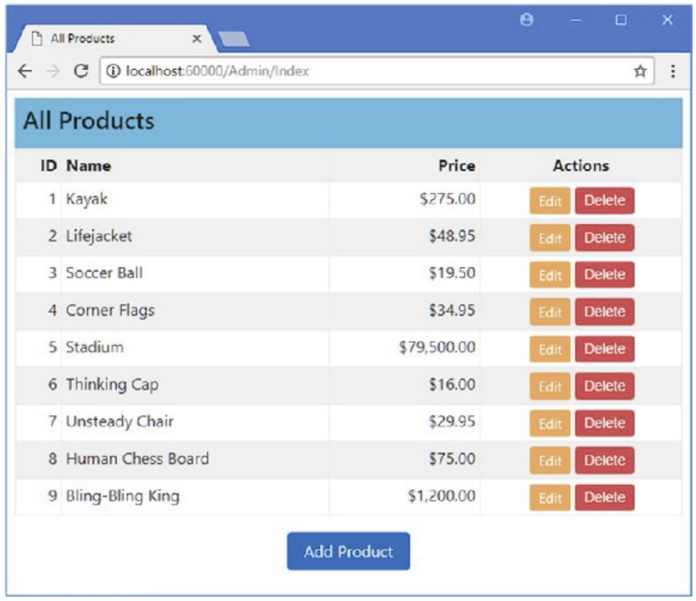


图11-3　显示商品列表

提　示

Edit按钮位于代码清单11-6中form元素的内部，因此Edit和Delete按钮彼此相邻，间距由

Bootstrap确定。Edit按钮将向服务器发送HTTP GET请求以获取商品的当前详细信息，这并不需要form元素。但是，由于Delete按钮将更改应用程序状态，需要使用HTTP POST请求，因此必须使用form元素。

11.2.3　编辑商品

为了提供创建和更新功能，添加图11-2所示的商品编辑页面。这项工作包括两部分：

- 显示一个允许管理员更改商品属性值的页面。
- 添加一个可以在提交时处理这些更改的操作方法。

1．创建**Edit**操作方法

代码清单11-7显示了添加到Admin控制器的

Edit操作方法，当用户单击Edit按钮时，它将接收浏览器发送的HTTP请求。

代码清单11-7　在Controllers文件夹下的AdminController.cs文件中添加Edit操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId));
    }
}
```

这个简单的方法会查找具有与productId参数对应的ID的商品，并作为视图模型对象传递给View方法。

测试**Edit**操作方法

我们想在Edit操作方法中测试两个功能。首先，当提供有效的id时，能够得到正确的商品。其次，当请求存储库中不存在的id时，不会得到任何商品。以下是添加到AdminControllerTests.cs类文件的测试方法：

```
...
[Fact]
public void Can_Edit_Product() {
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
```

```
    }.AsQueryable<Product>());
    // Arrange - create the controller
    AdminController target = new AdminController(mock.O
bject);

    // Act
    Product p1 = GetViewModel<Product>(target.Edit(1));
    Product p2 = GetViewModel<Product>(target.Edit(2));
    Product p3 = GetViewModel<Product>(target.Edit(3));

    // Assert
    Assert.Equal(1, p1.ProductID);
    Assert.Equal(2, p2.ProductID);
    Assert.Equal(3, p3.ProductID);
}

[Fact]
public void Cannot_Edit_Nonexistent_Product() {
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
    }.AsQueryable<Product>());

    // Arrange - create the controller
    AdminController target = new AdminController(mock.O
bject);

    // Act
    Product result = GetViewModel<Product>(target.Edit(
4));

    // Assert
```

```
    Assert.Null(result);
}
...
```

2．创建Edit视图

现在有了一个操作方法，可以创建一个视图来显示它。在Views/Admin文件夹中添加一个名为Edit.cshtml的Razor视图文件，并添加代码清单11-8所示的标记。

代码清单11-8　Views/Admin文件夹下的Edit.cshtml文件的内容

```
@model Product
@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
    </div>
```

```
    <div class="form-group">
        <label asp-for="Description"></label>
        <textarea asp-for="Description" class="form-con
trol"></textarea>
    </div>
    <div class="form-group">
        <label asp-for="Category"></label>
        <input asp-for="Category" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input asp-for="Price" class="form-control" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" type="submit">S
ave</button>
        <a asp-action="Index" class="btn btn-secondary"
>Cancel</a>
    </div>
</form>
```

Edit视图包含一个HTML表单，它使用标签助手生成大部分内容，包括设置form和a元素的目标，设置label元素的内容，以及为input和textarea元素生成name、id和value属性。

可以通过启动应用程序，导航到/Admin/Index

URL，然后单击其中一个商品的Edit按钮来查看视图生成的HTML，如图11-4所示。

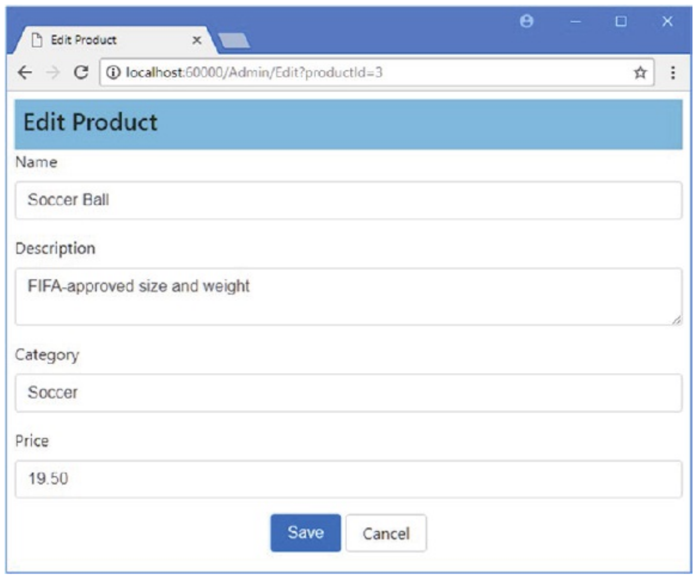


图11-4　显示商品的属性以进行编辑

提　示

为了简单起见，为ProductID属性使用隐藏的input元素。当Entity Framework Core存储新对象

时，由数据库生成ProductID的值并将其设置为主键，安全地更改它可能是一个复杂的过程。对于大多数应用程序，最简单的方法是防止用户更改值。

3．更新Product存储库

在处理编辑之前，还需要改进Product存储库，以便能够保存更改。首先，向IProductRepository接口添加一个新的方法，如代码清单11-9所示。

代码清单11-9　在Models文件夹下的IProductRepository.cs文件中添加一个新的方法

```
using System.Linq;

namespace SportsStore.Models {

    public interface IProductRepository {

        IQueryable<Product> Products { get; }

        void SaveProduct(Product product);
```

```
    }
}
```

然后，可以将这个新方法添加到由Entity Framework Core实现的存储库中，具体定义在EFProductRepository.cs文件中，如代码清单11-10所示。

代码清单11-10　在Models文件夹下的EFProductRepository.cs文件中实现这个新方法

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class EFProductRepository : IProductReposito
ry {
        private ApplicationDbContext context;

        public EFProductRepository(ApplicationDbContext
 ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.
Products;

        public void SaveProduct(Product product) {
```

```
            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                Product dbEntry = context.Products
                    .FirstOrDefault(p => p.ProductID ==
 product.ProductID);
                if (dbEntry != null) {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Descr
iption;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;

                }
            }
            context.SaveChanges();
        }
    }
}
```

如果ProductID为0，SaveChanges操作方法会将商品添加到存储库中；否则，就对数据库中的现有商品进行更新。

本章不会深入介绍Entity Framework Core的细节，因为如前所述，这本身就是一个较大的主题，而不是ASP.NET Core MVC的一部分。但是，

SaveProduct操作方法中有一些内容与MVC应用程序的设计有关。

当收到ProductID不为0的Product参数时，需要执行更新。可以通过以下步骤来实现：从存储库中获取具有相同ProductID的Product对象，并更新每个属性，使它们与参数对象匹配。

这样做是因为Entity Framework Core会跟踪从数据库创建的对象。传递给SaveChanges操作方法的对象是由MVC模型绑定功能创建的，这意味着Entity Framework Core对新的Product对象一无所知，在修改数据库时不会对数据库应用更新。有很多方法可以解决这个问题，这里采用最简单的方法，即找到Entity Framework Core跟踪的相应对象，并明确地更新它们。

在IProductRepository接口中添加新方法破坏了

已在第8章中创建的虚拟存储库类FakeProductRepository。可使用虚拟存储库启动开发过程，并演示如何使用服务来无缝替换接口实现，而无须修改依赖它们的组件。现在不再需要虚拟存储库，在代码清单11-11中，可以看到已从类声明中删除了接口，因此不必在添加存储库功能时继续修改该类。

代码清单11-11　删除Models文件夹下的FakeProductRepository.cs文件中的接口

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class FakeProductRepository /* : IProductRep
ository */ {

        public IQueryable<Product> Products => new List
<Product> {
            new Product { Name = "Football", Price = 25
 },
            new Product { Name = "Surf board", Price =
179 },
            new Product { Name = "Running shoes", Price
```

```
= 95 }
        }.AsQueryable<Product>();
    }
}
```

4．处理编辑POST请求

现在可以在Admin控制器中实现Edit操作方法的重载，从而在管理员单击Save按钮时处理POST请求。代码清单11-12显示了新的Edit操作方法。

代码清单11-12　在Controllers文件夹下的AdminController.cs文件中定义新的Edit操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }
```

```
        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId));

        [HttpPost]
        public IActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name}
has been saved";
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the da
ta values
                return View(product);
            }
        }
    }
}
```

可通过检查模型绑定过程是否能够读取ModelState.IsValid属性的值来验证用户提交的数据。如果一切正常，就将更改保存到存储库并将用户重定向到Index操作方法，以便他们看到修改后的商品列表。如果数据有问题，会再次渲染默认视

图，以便用户进行更正。

保存存储库中的更改后，使用temp data功能存储消息，temp data功能是ASP.NET Core会话状态功能的一部分。这是键/值字典，类似于之前使用的会话数据和view bag功能，与会话数据的主要区别在于临时数据在读取之前一直存在。

在这种情况下不能使用ViewBag，因为ViewBag在控制器和视图之间传递数据，并且数据的保存时间不能超过当前的HTTP请求。编辑成功后，浏览器将重定向到新的URL，因此ViewBag数据将丢失。可以使用会话数据功能，但是在明确删除之前，消息将一直存在，这里并不想这样做。

因此，temp data功能是最合适的。数据仅限于单个用户的会话（因此用户看不到彼此的临时数据）并且会持续足够长的时间以供用户读取。这里

将读取重定向用户的操作方法呈现的视图中的数据。

单元测试——编辑提交的数据

对于Edit操作方法处理POST请求的过程，需要确保将作为方法参数接收的Product对象的有效更新传递到Product存储库以进行保存。这里还需要检查无效更新（在模型验证错误的情况下）不会传递到存储库。以下是添加到AdminControllerTests.cs文件中的测试方法：

```
...
[Fact]
public void Can_Save_Valid_Changes() {
    // Arrange - create mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    // Arrange - create mock temp data
    Mock<ITempDataDictionary> tempData = new Mock<ITemp
DataDictionary>();
    // Arrange - create the controller
```

```
    AdminController target = new AdminController(mock.O
bject) {
        TempData = tempData.Object
    };
    // Arrange - create a product
    Product product = new Product { Name = "Test" };

    // Act - try to save the product
    IActionResult result = target.Edit(product);
    // Assert - check that the repository was called
    mock.Verify(m => m.SaveProduct(product));
    // Assert - check the result type is a redirection
    Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Index", (result as RedirectToActionRe
sult).ActionName);
}

[Fact]
public void Cannot_Save_Invalid_Changes() {
    // Arrange - create mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    // Arrange - create the controller
    AdminController target = new AdminController(mock.O
bject);
    // Arrange - create a product
    Product product = new Product { Name = "Test" };
    // Arrange - add an error to the model state
    target.ModelState.AddModelError("error", "error");

    // Act - try to save the product
    IActionResult result = target.Edit(product);
    // Assert - check that the repository was not calle
d
    mock.Verify(m => m.SaveProduct(It.IsAny<Product>())
, Times.Never());
```

```
    // Assert - check the method result type
    Assert.IsType<ViewResult>(result);
}
```

...

5．显示确认消息

在_AdminLayout.cshtml布局文件中处理使用TempData存储的消息，如代码清单11-13所示。通过处理模板中的消息，可以在任何使用模板的视图中创建消息，而无须创建其他Razor表达式。

代码清单11-13　处理_AdminLayout.cshtml文件中的ViewBag消息

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
```

```
    <title>@ViewBag.Title</title>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></d
iv>
    @if (TempData["message"] != null) {
        <div class="alert alert-success">@TempData["mes
sage"]</div>
    }
    @RenderBody()
</body>
</html>
```

提　示

在模板中处理消息的好处是，在保存所做更改后的所有页面上，用户都能看到它们。目前，将它们返回商品列表中，但可以更改工作流以呈现其他视图，用户仍将看到消息（只要下一个视图也使用相同的布局）。

现在已经准备好用于编辑商品的所有代码。要查看具体是如何工作的，请启动应用程序，导航到/Admin/Index URL，单击Edit按钮，然后进行更改。单击Save按钮，你将被重定向到/Admin/Index URL，并显示TempData消息，如图11-5所示。

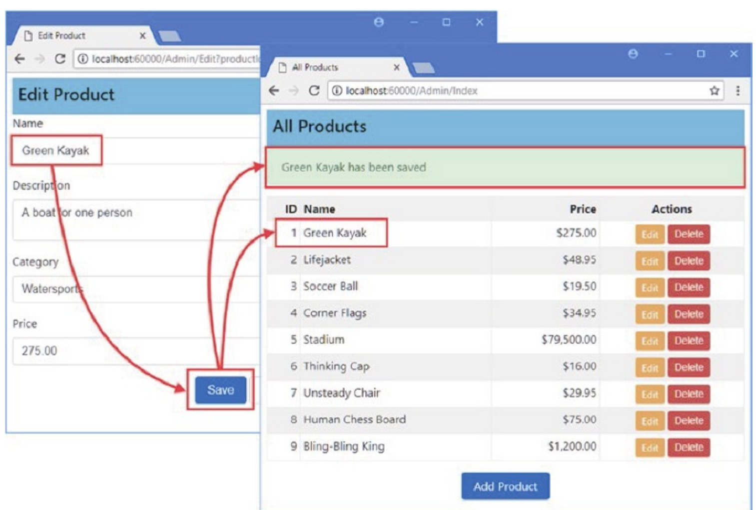


图11-5　编辑商品并查看TempData消息

如果重新加载商品列表视图，消息将消失，因为TempData在读取时会被删除。这很方便，因为作者不希望一直显示旧的信息。

6．添加模型验证

接下来需要向模型类添加验证规则。目前，管理员可以输入负的价格或空白描述，SportsStore应用程序会很乐意将这些数据存储在数据库中。坏数据是否会成功保留将取决于它们是否符合Entity Framework Core创建的SQL表中的约束，这对于大多数应用程序来说保护得还不够。为了防止错误的数据值，使用特性修饰Product类的属性，如代码清单11-14所示，就像在第10章中对Order类所做的那样。

代码清单11-14　在Models文件夹下的Product.cs文件中应用验证特性

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Product {
        public int ProductID { get; set; }
```

```
        [Required(ErrorMessage = "Please enter a produc
t name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a descri
ption")]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue,
            ErrorMessage = "Please enter a positive pri
ce")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a cate
gory")]
        public string Category { get; set; }
    }
}
```

第10章使用标签助手在表单顶部显示验证错误的摘要。在本例中，将使用类似的方法，但是将在Edit视图中的各个表单元素旁显示错误消息，如代码清单11-15所示。

代码清单11-15　在Views/Admin文件夹下的Edit.cshtml文件中添加验证错误元素

```
@model Product
```

```
@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />
    <div class="form-group">
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="tex
t-danger"></span></div>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Description"></label>
        <div><span asp-validation-for="Description" cla
ss="text-danger"></span></div>
        <textarea asp-for="Description" class="form-con
trol"></textarea>
    </div>
    <div class="form-group">
        <label asp-for="Category"></label>
        <div><span asp-validation-for="Category" class=
"text-danger"></span></div>
        <input asp-for="Category" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <div><span asp-validation-for="Price" class="te
xt-danger"></span></div>
        <input asp-for="Price" class="form-control" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" type="submit">S
ave</button>
```

```
        <a asp-action="Index" class="btn btn-secondary"
>Cancel</a>
    </div>
</form>
```

当应用于span元素时，为asp-validation-for属性应用标签助手，从而当存在任何验证问题时，为指定的属性添加验证错误消息。

标签助手将向span元素插入一条错误消息，并向span元素添加input-validation-error类，这样可以很容易地将CSS样式应用于错误消息元素，如代码清单11-16所示。

代码清单11-16　将CSS添加到Views/Shared文件夹下的_AdminLayout.cshtml文件中

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
```

```
    <style>
        .input-validation-error { border-color: red; ba
ckground-color: #fee ; }
    </style>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></d
iv>
    @if (TempData["message"] != null) {
        <div class="alert alert-success mt-1">@TempData
["message"]</div>
    }
    @RenderBody()
</body>
</html>
```

定义的CSS样式将选择作为input-validation-error类成员的元素，并应用红色边框和背景颜色。

提　示

在使用Bootstrap这样的CSS库时，显式地设置

样式会在应用内容时导致主题不一致。第27章将展示一种替代方法，该方法使用JavaScript代码将Bootstrap类应用于具有验证错误的元素，这虽然使得所有内容保持一致，但也更复杂。

可以在视图中的任何位置应用验证消息标签助手，但常规（并且合理的）做法是将标签助手放在问题元素附近的某个位置，以向用户提供一些上下文信息。图11-6显示了验证消息和提示，可以通过运行应用程序、编辑商品和提交无效数据来查看这些验证消息和提示。

7. 启用客户端验证

目前，仅当管理员用户向服务器提交编辑时才应用数据验证，但是大多数用户希望如果输入的数

据存在问题，能够立即得到反馈。这就是开发人员经常想要执行客户端验证的原因，即使用JavaScript在浏览器中检查数据。MVC应用程序可以基于应用于域模型类的数据注解来执行客户端验证。

第一步是向应用程序添加提供客户端功能的JavaScript库，这在bower.json文件中完成，如代码清单11-17所示。

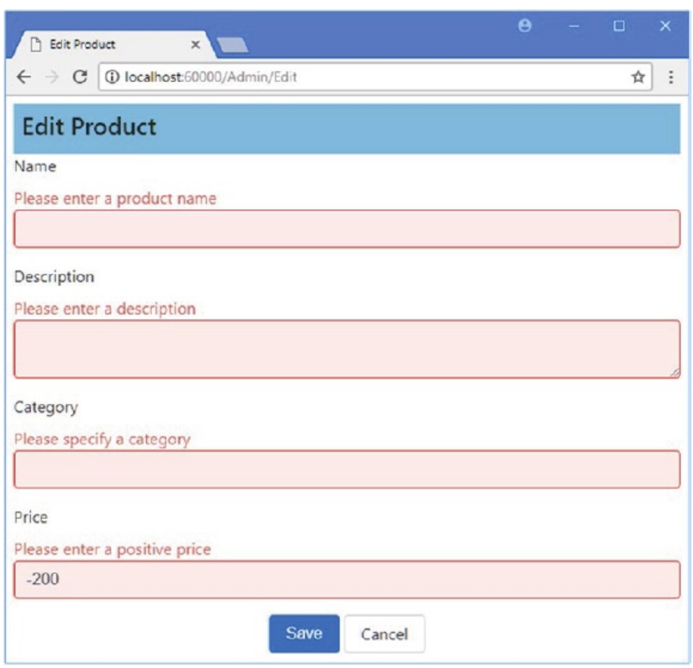


图11-6　编辑商品时执行数据验证

代码清单11-17　在bower.json文件中添加JavaScript包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

客户端验证建立在流行的jQuery库之上，简化了浏览器的DOM API。下一步是将JavaScript文件添加到布局中，以便在使用SportsStore管理功能时加载它们，如代码清单11-18所示。

代码清单11-18　将验证库添加到Views/Shared文件夹下的_AdminLayout.cshtml文件中

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" asp-href-include="lib/bootst
```

```
rap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
    <style>
        .input-validation-error { border-color: red; ba
ckground-color: #fee ; }
    </style>
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <script src="/lib/jquery-validation/dist/jquery.val
idate.min.js"></script>
    <script
      src="/lib/jquery-validation-unobtrusive/jquery.va
lidate.unobtrusive.min.js">
    </script>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></d
iv>
    @if (TempData["message"] != null) {
        <div class="alert alert-success mt-1">@TempData
["message"]</div>
    }
    @RenderBody()
</body>
</html>
```

启用客户端验证不会导致任何可视化更改，但是由C#模型类属性指定的约束将在浏览器中强制执行，从而阻止用户提交包含错误数据的表单，并在有问题时立即提供反馈。有关详细信息，请参见第

27章。

11.2.4 创建新的商品

接下来，将实现Create操作方法，这个操作方法在主商品列表页面中由Add Product链接指定的。这将允许管理员将新的商品添加到商品目录中。添加创建新商品的功能需要对应用程序进行一点小的更改。这是一个很好的例子，展示了结构良好的MVC应用程序具有的强大功能和灵活性。首先，将Create操作方法（如代码清单11-19所示）添加到Admin控制器中。

代码清单11-19 将Create操作方法添加到Controllers文件夹下的AdminController.cs文件中

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {
```

```
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId));

        [HttpPost]
        public IActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name}
has been saved";
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the da
ta values
                return View(product);
            }
        }

        public ViewResult Create() => View("Edit", new
Product());
    }
}
```

Create操作方法不会渲染默认视图，而是指定应使用Edit视图。一个操作方法完全可以使用通常与另一个操作方法关联的视图。在这种情况下，可以提供一个新的Product对象作为视图模型，以便使用空字段填充Edit视图。

注　意

这里没有为Create操作方法添加单元测试。这样做只会测试ASP.NET Core MVC处理来自操作方法的结果的能力，这是理所当然的（除非怀疑存在缺陷，否则通常不会为框架功能编写测试）。

这是唯一需要执行的更改，因为Edit操作方法已设置为从模型绑定系统接收Product对象并将它们存储在数据库中。可以通过启动应用程序，导航到/Admin/Index，单击Add Product按钮，然后填充并提交表单来测试此功能。你在表单中指定的详细信息将用于在数据库中创建新的商品，然后新的商品将显示在列表中，如图11-7所示。

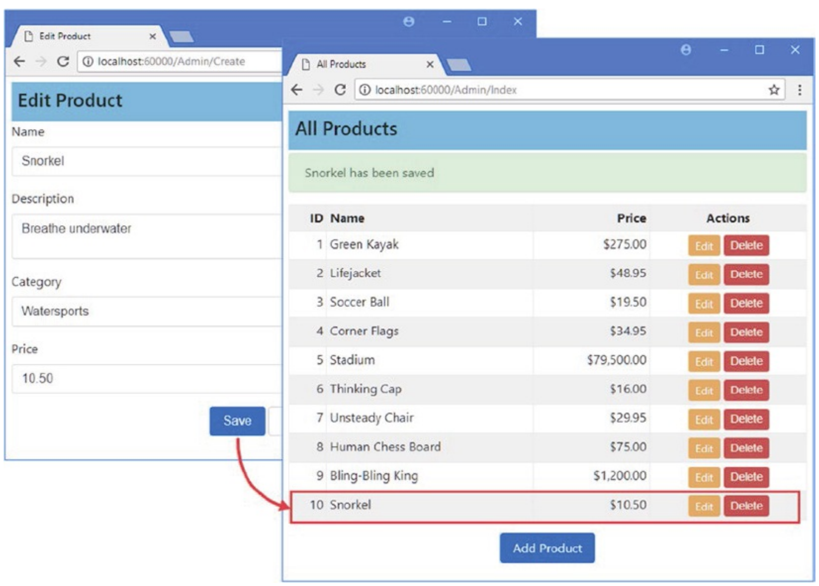


图11-7　将新的商品添加到目录中

11.2.5　删除商品

添加删除商品的功能也很简单。第一步是向IProductRepository接口添加一个新的方法，如代码清单11-20所示。

代码清单11-20　在Models文件夹下的IProductRepository.cs文件中添加商品删除方法

```
using System.Linq;

namespace SportsStore.Models {

    public interface IProductRepository {

        IQueryable<Product> Products { get; }

        void SaveProduct(Product product);

        Product DeleteProduct(int productID);
    }
}
```

接下来，在Entity Framework Core存储库类EFProductRepository中实现这个方法，如代码清单11-21所示。

代码清单11-21　在Models文件夹下的EFProductRepository.cs文

件中实现商品删除功能

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class EFProductRepository : IProductReposito
ry {
        private ApplicationDbContext context;

        public EFProductRepository(ApplicationDbContext
 ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.
Products;

        public void SaveProduct(Product product) {
            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                Product dbEntry = context.Products
                    .FirstOrDefault(p => p.ProductID ==
 product.ProductID);
                if (dbEntry != null) {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Descr
iption;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category
;
                }
```

```
            }
            context.SaveChanges();
        }
        public Product DeleteProduct(int productID) {
            Product dbEntry = context.Products
                .FirstOrDefault(p => p.ProductID == pro
ductID);
            if (dbEntry != null) {
                context.Products.Remove(dbEntry);
                context.SaveChanges();
            }
            return dbEntry;
        }
    }
}
```

最后一步是在Admin控制器中实现Delete操作方法。这个操作方法应仅支持POST请求，因为删除对象不是幂等操作。正如在第16章中解释的那样，浏览器和缓存可以在未经用户明确同意的情况下自由发出GET请求，因此必须小心避免因GET请求而进行更改的情况。代码清单11-22显示了新的Delete操作方法。

代码清单11-22　在Controllers文件夹下的AdminController.cs文件中添加Delete操作方法

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId));

        [HttpPost]
        public IActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name}
has been saved";
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the da
ta values
                return View(product);
            }
        }
```

```
        public IActionResult Create() => View("Edit", new Product());

        [HttpPost]
        public IActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteProduct(productId);
            if (deletedProduct != null) {
                TempData["message"] = $"{deletedProduct.Name} was deleted";
            }
            return RedirectToAction("Index");
        }
    }
}
```

单元测试——删除商品

下面测试 Delete 操作方法的基本功能：当把有效的 ProductID 作为参数传递时，调用存储库的DeleteProduct方法，并传递正确的ProductID值进行删除。下面是添加到AdminControllerTests.cs文件的测试：

```
...
```

```
[Fact]
public void Can_Delete_Valid_Products() {
    // Arrange - create a Product
    Product prod = new Product { ProductID = 2, Name =
"Test" };

    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRe
pository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        prod,
        new Product {ProductID = 3, Name = "P3"},
    }.AsQueryable<Product>());

    // Arrange - create the controller
    AdminController target = new AdminController(mock.O
bject);

    // Act - delete the product
    target.Delete(prod.ProductID);

    // Assert - ensure that the repository delete metho
d was
    // called with the correct Product
    mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
...
```

可以通过启动应用程序，导航

到/Admin/Index，然后单击商品列表页面中的其中一个Delete按钮来查看删除功能，如图11-8所示。当从目录中删除商品时，这里使用TempData变量显示消息。

注　意

如果删除先前已创建订单的商品，则会发现错误。当Order对象存储在数据库中时，它将被转换为数据表中的一条记录，其中包含与之关联的Product对象的引用，称为外键关系。这意味着默认情况下，如果已为商品创建订单，数据库将不允许删除Product对象，因为这样做会在数据库中导致不一致。有许多方法可用来解决此问题，包括在删除

商品时自动删除与之相关的订单对象或者更改商品和订单对象之间的关系。详细信息请参阅Entity Framework Core文档。

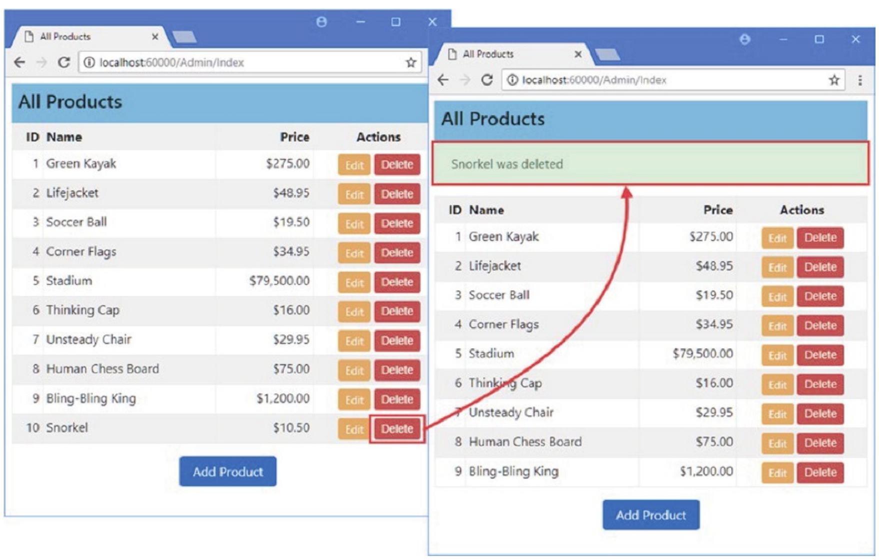


图11-8　从目录中删除商品

11.3　小结

本章介绍了管理功能，并展示了如何实现CRUD操作，从而允许管理员在存储库中创建、读

取、更新和删除商品，并将订单标记为已发货。下一章将展示如何保护管理功能，使其不对所有用户公开，并将SportsStore应用程序部署到生产环境中。

第12章　SportsStore的安全和部署

在上一章中，添加了SportsStore应用程序的管理功能，如果按原样部署应用程序，可能没人注意到任何人都可以修改商品目录。他们需要知道的是，任何人使用URL/Admin/Index和/Order/List都可以执行用管理功能。本章将展示如何通过密码保护来阻止任意用户使用管理功能。一旦具备了安全措施，就将展示如何准备和部署SportsStore应用程序到生产环境中。

12.1　保护管理功能

ASP.NET Core Identity系统提供了身份验证和授权，该系统可以整合到ASP.NET Core平台和MVC应用程序中。在接下来的内容中，将创建基本

的安全设置，允许名为Admin的用户对应用程序中的管理功能进行身份验证和访问。ASP.NET Core Identity提供了许多用于验证用户身份和授权访问应用程序特性和数据的功能，可以在第28～30章中找到更详细的信息，其中将展示如何创建和管理用户账户、如何使用角色和策略，以及如何支持来自Microsoft、Google、Facebook和Twitter等第三方的身份验证。但是，在本章中，目标是获得足够的功能以防止客户访问SportsStore应用程序的敏感部分，并在此过程中，让你了解在MVC应用中如何使用身份验证和授权。

12.1.1　创建身份标识数据库

ASP.NET Core Identity系统具有无限的可配置性和可扩展性，并支持多种用户数据存储方式。这里将使用最常见的使用Entity Framework Core访问的Microsoft SQL Server存储数据。

1．创建上下文类

这里需要创建一个数据库上下文文件，以充当数据库与提供访问权限的Identity模型对象之间的桥梁。在Model文件夹中添加一个名为AppIdentityDbContext.cs的类文件，并用它定义代码清单12-1所示的类。

注　意

你可能习惯于将包添加到项目中以获得安全性等其他功能。但是，随着ASP.NET Core 2的发布，身份标识所需的NuGet包已经包含在项目中，也就是作为项目模板的一部分添加到SportsStore.csproj文件中的元数据包。

代码清单12-1　Models文件夹下的AppIdentityDbContext.cs文件的内容

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore
;
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {

    public class AppIdentityDbContext : IdentityDbConte
xt<IdentityUser> {

        public AppIdentityDbContext(DbContextOptions<Ap
pIdentityDbContext> options)
            : base(options) { }
    }
}
```

AppIdentityDbContext类派生自IdentityDbContext，后者为Entity Framework Core提供特定于身份标识的功能。对于type参数，使用IdentityUser类，它是用于表示用户的内置类。第28章将演示如何使用可扩展的自定义类来添加应用程

序用户的额外信息。

2．定义连接字符串

下一步是定义数据库的连接字符串。在代码清单12-2中，可以看到对SportsStore项目的appsettings.json文件所做的补充，格式与在第8章中为产品数据库定义的连接字符串相同。

代码清单12-2　在appsettings.json文件中定义连接字符串

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": " Server=(localdb)\\MSSQLLoca
lDB;Database=SportsStore;Trusted_
                            Connection=True;MultipleAct
iveResultSets=true"
    },
    "SportStoreIdentity": {
      "ConnectionString": " Server=(localdb)\\MSSQLLoca
lDB;Database=Identity;Trusted_
                            Connection=True;MultipleAct
iveResultSets=true"
    }
  }
}
```

请记住，连接字符串必须在appsettings.json文件中的单个连续行中定义（这里由于受版面宽度的限制换行显示）。代码清单12-2定义了一个名为SportsStoreIdentity的连接字符串，其中指定了名为Identity的LocalDB数据库。

3．配置应用程序

与其他ASP.NET Core功能一样，ASP.NET Core Identity在Start类中配置。代码清单12-3显示了使用上面定义的上下文类和连接字符串在SportsStore项目中为设置ASP.NET Core Identity所做的补充。

代码清单12-3　在Startup.cs文件中配置身份标识

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<IdentityUser, Identity
Role>()
                .AddEntityFrameworkStores<AppIdentityDb
Context>()
                .AddDefaultTokenProviders();
```

```
            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddScoped<Cart>(sp => SessionCart.
GetCart(sp));
            services.AddSingleton<IHttpContextAccessor,
 HttpContextAccessor>();
            services.AddTransient<IOrderRepository, EFO
rderRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseAuthentication();
            app.UseMvc(routes => {

                // ...routes omitted for brevity...

            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

在ConfigureServices方法中，扩展Entity Framework Core配置以注册上下文类，并使用

AddIdentity方法使用内置类来设置ASP.NET Core Identity服务以表示用户和角色。在Configure方法中，调用UseAuthentication方法来设置组件，用于拦截请求和响应以实现安全策略。

4．创建和应用数据库迁移

基本配置已就位，现在可以使用Entity Framework Core迁移功能来定义数据库架构并将其应用于数据库。打开新的命令提示符或PowerShell窗口，并在SportsStore项目文件夹中运行以下命令，为Identity数据库创建新的迁移：

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

与以前的数据库命令的重要区别在于，使用-context参数来指定与要使用的数据库关联的上下文类的名称，这里是AppIdentityDbContext。当应用程

序中有多个数据库时，确保使用正确的上下文类非常重要。

一旦Entity Framework Core生成了初始迁移，就运行以下命令来创建数据库并运行迁移命令：

```
dotnet ef database update --context AppIdentityDbContext
```

结果是生成名为Identity的LocalDB数据库，该数据库可以使用Visual Studio SQL Server对象资源管理器进行查看。

5．定义种子数据

可通过在应用程序启动时对数据库进行填充来显式创建Admin用户。在Models文件夹中添加一个名为IdentitySeedData.cs的类文件，并定义代码清单12-4所示的静态类。

代码清单12-4　Models文件夹下的IdentitySeedData.cs文件的内

容

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123
$";

        public static async void EnsurePopulated(IAppli
cationBuilder app) {

            UserManager<IdentityUser> userManager = app
.ApplicationServices
                .GetRequiredService<UserManager<Identit
yUser>>();

            IdentityUser user = await userManager.FindB
yIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                await userManager.CreateAsync(user, adm
inPassword);
            }
        }
    }
}
```

以上代码使用了UserManager <T>类，它由

ASP.NET Core Identity作为服务提供，用于管理用户，如第28章所述。数据库用来查找管理员账户，管理员账户是使用密码Secret123 $创建的。请勿在此例中更改硬编码的密码，因为ASP.NET Core Identity具有验证策略，会要求密码包含数字和指定范围的字符。有关如何更改验证设置的详细信息，请参阅第28章。

警　告

在开发过程中经常需要对管理员账户的详细信息进行硬编码，以便在部署应用程序后立即登录并开始管理。执行此操作时，必须更改已创建账户的密码。有关如何使用身份标识更改密码的详细信

息，请参阅第28章。

为了确保在应用程序启动时填充Identity数据库，将代码清单12-5所示的语句添加到Startup类的Configure方法中。

代码清单12-5　在SportsStore文件夹下的Startup.cs文件中填充Identity数据库

```
...
public void Configure(IApplicationBuilder app, IHosting
Environment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseMvc(routes => {
        // ...routes omitted for brevity...

    });
    SeedData.EnsurePopulated(app);
    IdentitySeedData.EnsurePopulated(app);
}
...
```

12.1.2　应用基本授权策略

既然已经配置了ASP.NET Core Identity，就可以将授权策略应用于想要保护的应用程序。可使用最基本的授权策略，允许任何经过身份验证的用户访问。虽然这在实际的应用程序中是一种有用的策略，但还可以选择创建更精细的授权控制（如第28～30章所述），但由于SportsStore应用程序只有一个用户，因此能够区分匿名请求和经过身份验证的请求就足够了。

Authorize特性用于限制对操作方法的访问，在代码清单12-6中，可以看到已使用Authorize特性来保护对Order控制器中管理操作的访问。

代码清单12-6　限制OrderController.cs文件中的访问权限

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;
```

```
namespace SportsStore.Controllers {

    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;

        public OrderController(IOrderRepository repoSer
vice, Cart cartService) {
            repository = repoService;
            cart = cartService;
        }

        [Authorize]
        public ViewResult List() =>
            View(repository.Orders.Where(o => !o.Shippe
d));

        [HttpPost]
        [Authorize]
        public IActionResult MarkShipped(int orderID) {
            Order order = repository.Orders
                .FirstOrDefault(o => o.OrderID == order
ID);
            if (order != null) {
                order.Shipped = true;
                repository.SaveOrder(order);
            }
            return RedirectToAction(nameof(List));
        }
        public ViewResult Checkout() => View(new Order(
));

        [HttpPost]
        public IActionResult Checkout(Order order) {
            if (cart.Lines.Count() == 0) {
```

```
                ModelState.AddModelError("", "Sorry, yo
ur cart is empty!");
            }
            if (ModelState.IsValid) {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Complete
d));
            } else {
                return View(order);
            }
        }

        public ViewResult Completed() {
            cart.Clear();
            return View();
        }
    }
}
```

由于不想阻止未经身份验证的用户访问Order控制器中的其他操作方法，因此仅将Authorize特性应用于List和MarkShipped操作方法。如果要保护Admin控制器定义的所有操作方法，那么可以通过将Authorize特性应用于控制器类来实现这一点，控制器类会将授权策略应用于它包含的所有操作方法，如代码清单12-7所示。

代码清单12-7　限制AdminController.cs文件中的访问权限

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == pro
ductId));
        [HttpPost]
        public IActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name}
has been saved";
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the da
```

```
ta values
                return View(product);
            }
        }

        public ViewResult Create() => View("Edit", new
Product());

        [HttpPost]
        public IActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteP
roduct(productId);
            if (deletedProduct != null) {
                TempData["message"] = $"{deletedProduct
.Name} was deleted";
            }
            return RedirectToAction("Index");
        }
    }
}
```

12.1.3 创建账户控制器和视图

当未经身份验证的用户发送需要身份验证的请求时，他们会被重定向到/Account/Login URL，应用程序将提示用户需要提供凭据。在准备过程中，可添加视图模型来表示用户的凭据，方法是将名为LoginModel.cs的类文件添加到Models/ViewModels

文件夹中，并用它定义代码清单12-8所示的类。

代码清单12-8　Models/ViewModels文件夹下的LoginModel.cs文件的内容

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels {

    public class LoginModel {

        [Required]
        public string Name { get; set; }

        [Required]
        [UIHint("password")]
        public string Password { get; set; }

        public string ReturnUrl { get; set; } = "/";
    }
}
```

Name和Password属性已使用Required特性进行修饰，该特性使用模型验证来确保必须为其提供值。Password属性已使用UIHint特性进行了修饰，因此，当在Razor视图Login中的input元素上使用asp-for属性时，标签助手会将type属性设置为

password；这样，用户输入的文本在屏幕上将是不可见的。第24章将描述UIHint特性的使用方法。

接下来，将一个名为AccountController.cs的类文件添加到Controllers文件夹中，并用它定义代码清单12-9所示的控制器，以响应对/Account/Login URL的请求。

代码清单12-9　Controllers文件夹下的AccountController.cs文件的内容

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManag
er;

        public AccountController(UserManager<IdentityUs
er> userMgr,
```

```
                SignInManager<IdentityUser> signInMgr)
{
            userManager = userMgr;
            signInManager = signInMgr;
        }

        [AllowAnonymous]
        public ViewResult Login(string returnUrl) {
            return View(new LoginModel {
                ReturnUrl = returnUrl
            });
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginMod
el loginModel) {
            if (ModelState.IsValid) {
                IdentityUser user =
                    await userManager.FindByNameAsync(l
oginModel.Name);
                if (user != null) {
                    await signInManager.SignOutAsync();
                    if ((await signInManager.PasswordSi
gnInAsync(user,
                            loginModel.Password, false,
 false)).Succeeded) {
                        return Redirect(loginModel?.Ret
urnUrl ?? "/Admin/Index");
                    }
                }
            }
            ModelState.AddModelError("", "Invalid name
or password");
            return View(loginModel);
```

```
        }

        public async Task<RedirectResult> Logout(string
 returnUrl = "/") {
            await signInManager.SignOutAsync();
            return Redirect(returnUrl);
        }
    }
}
```

当用户被重定向到/Account/Login URL时，Login操作方法的GET版本将呈现页面的默认视图，并提供一个视图模型对象，该对象包括一个URL。如果身份验证请求成功，浏览器将重定向到这个URL。

身份验证凭据将被提交到Login操作方法的POST版本，该操作方法使用了UserManager<IdentityUser>和SignInManager<IdentityUser>服务，这两个服务是通过控制器的构造函数接收的，用于对用户进行身份验证并登录系统。第28～30章将解释这些类是如

何工作的，但是现在知道以下内容就足够了：如果身份验证失败，那么将创建模型验证错误并呈现默认视图；但是如果身份验证成功，那么会在提示输入用户凭据之前将用户重定向到他们要访问的URL。

警　告

　　一般情况下，使用客户端数据验证是一个好主意。这可以减轻服务器的工作量，并为用户提示有关他们所提供数据的即时反馈。但是，你不应该在客户端执行身份验证，因为这通常涉及将有效凭据发送到客户端，以便它们检查用户输入的用户名和密码，或者至少信任客户端关于他们是否已成功通

过身份验证的报告。身份验证应始终在服务器端完成。

为了向Login操作方法提供渲染视图，创建Views/Account文件夹并在其中添加一个名为Login.cshtml的Razor视图文件，其内容如代码清单12-10所示。

代码清单12-10　Views/Account文件夹下的Login.cshtml文件的内容

```
@model LoginModel
@{
    ViewBag.Title = "Log In";
    Layout = "_AdminLayout";
}

<div class="text-danger" asp-validation-summary="All"><
/div>

<form asp-action="Login" asp-controller="Account" metho
d="post">
    <input type="hidden" asp-for="ReturnUrl" />
    <div class="form-group">
```

```
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="tex
t-danger"></span></div>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <div><span asp-validation-for="Password" class=
"text-danger"></span></div>
        <input asp-for="Password" class="form-control"
/>
    </div>
    <button class="btn btn-primary" type="submit">Log I
n</button>
</form>
```

最后一步是更改共享的管理布局，可通过添加一个按钮，向Logout操作发送请求使当前用户注销登录，如代码清单12-11所示。这是一个非常有用的功能，可以让你更轻松地测试应用程序。如果没有这个功能，那么需要清除浏览器的cookie才能返回未经身份验证的状态。

代码清单12-11　在_AdminLayout.cshtml文件中添加Log Out按钮

```
<!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
    <style>
        .input-validation-error {
            border-color: red;
            background-color: #fee;
        }
    </style>
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <script src="/lib/jquery-validation/dist/jquery.val
idate.min.js"></script>
    <script
      src="/lib/jquery-validation-unobtrusive/jquery.va
lidate.unobtrusive.min.js">
    </script>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2 row">
        <div class="col">
            <h4>@ViewBag.Title</h4>
        </div>
        <div class="col-2">
            <a class="btn btn-sm btn-primary"
               asp-action="Logout" asp-controller="Acco
unt">Log Out</a>
        </div>
    </div>
    @if (TempData["message"] != null) {
        <div class="alert alert-success mt-1">@TempData
["message"]</div>
```

```
    }
    @RenderBody()
</body>
</html>
```

12.1.4　测试安全策略

一切就绪后，你就可以通过启动应用程序并请求/Admin/Index 来测试安全策略了。由于你目前尚未经过身份验证，并且尝试访问需要授权的操作，因此你的浏览器将被重定向到/Account/Login URL。输入Admin和Secret123$（作为用户名和密码）并提交表单。Account控制器将检查提供的凭据以及添加到Identity数据库中的种子数据，如果输入了正确的信息，就会对你进行身份验证并将浏览器重定向回/Account/Login URL，现在就可以进行访问了。图12-1说明了上述过程。

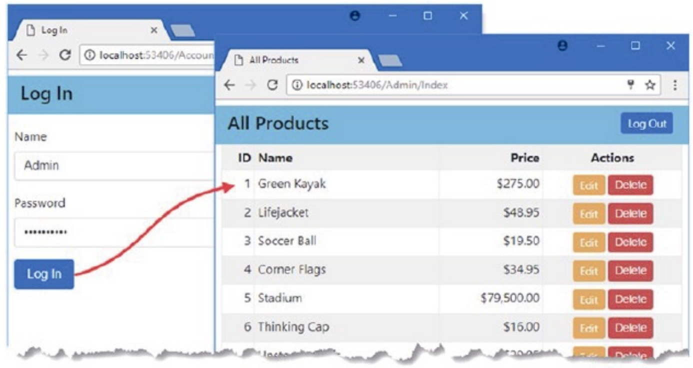


图12-1　管理验证/授权过程

12.2　部署应用程序

SportsStore应用程序的所有特性和功能都已到位，因此是时候准备将应用程序部署到生产环境中了。ASP.NET Core MVC应用程序有很多托管选项，本章使用的是Microsoft Azure平台，选择它是因为它来自微软，而且提供免费账户，这意味着即使不想将Microsoft Azure用于自己的项目，也可以一直使用SportsStore示例。

注　意

你需要一个Microsoft Azure账户。如果还没有，可以在Microsoft官网上申请免费账户。

12.2.1　创建数据库

我们从创建SportsStore应用程序将在生产环境中使用的数据库开始。这个操作可以在Visual Studio部署过程中执行，但这有点像鸡与蛋的关系，由于需要在部署之前知道数据库的连接字符串，因此需要先完成创建数据库的过程。

警　告

微软会向Microsoft Azure添加新功能并修改现有功能，因此Microsoft Azure门户网站经常更新。在撰写这些内容时，本节中的说明是准确的，但是在你阅读本书时，所需的步骤可能会略有改变。基本方法应该仍然相同，但数据字段的名称和步骤的确切顺序可能需要通过一些实验才能获得正确的结果。

最简单的方法是使用Microsoft Azure账户登录Azure门户网站并手动创建数据库。登录后，选择SQL Databases resource类别，然后单击Add按钮以创建新的数据库。

对于第一个数据库，输入名称products。单击Configure Required Settings链接，然后单击Create a New Server链接。输入新的服务器名称（在Microsoft Azure中必须是唯一的），并选择数据库管理员的用户名和密码。此处输入服务器名称sportsstorecore2db，数据库管理员的用户名为sportsstoreadmin，密码为Secret123$。你需要使用不同的服务器名称，建议使用更复杂的密码。选择数据库的位置，单击Select按钮以关闭选项，然后单击Create按钮以创建数据库。Microsoft Azure将花费几分钟时间来执行创建过程，之后它将出现在SQL Databases resource类别中。

创建另一个SQL服务器，这次输入名称**identity**。可以使用刚刚创建的数据库服务器，而不需要创建新的数据库服务器。结果是Microsoft Azure托管的两个SQL Server数据库，其详细信息如

表12-1所示。你将拥有不同的数据库服务器名称，理想情况下，你应该使用更健壮的密码。

表**12-1　SportsStore**应用程序的**Microsoft Azure**数据库

数据库名称	数据库服务器名称	数据库管理员	密码
products	sportsstorecore2db	sportsstoreadmin	Secret123$
identity	sportsstorecore2db	sportsstoreadmin	Secret123$

1．配置防火墙访问权限

需要按照数据库架构填充数据，最简单的方法是打开Microsoft Azure防火墙，以便从开发机器运行Entity Framework Core命令。

选择SQL Databases resource类别中的任意数据库，单击Tools按钮，然后单击Open in Visual Studio链接。现在单击Configure Your Firewall链接，单击

Add Client IP按钮，然后单击Save按钮，这将允许你从当前IP地址访问数据库服务器并执行配置命令（可以通过单击Open In Visual Studio按钮来检查数据库架构，这将打开Visual Studio并使用SQL Server对象资源管理器来检查数据库）。

2．获取连接字符串

接下来需要新的数据库连接字符串。当通过Show Database Connection Strings链接单击SQL Databases resource类别中的数据库时，Microsoft Azure会为不同的开发平台提供连接字符串，这是.NET应用程序所需的ADO.NET字符串。以下是Microsoft Azure门户为products数据库提供的连接字符串：

```
Server=tcp:sportsstorecore2db.database.windows.net,1433
;Initial Catalog=products;Persist
Security Info=False;User ID={your_username};Password={y
our_password};MultipleActiveResult
Sets=True;Encrypt=True;TrustServerCertificate=False;Con
```

```
nection Timeout=30;
```

你将看到不同的配置选项，具体取决于Microsoft Azure如何配置数据库。请注意，用户名和密码有占位符，这里已经用粗体标记了，当使用连接字符串配置应用程序时必须更改这些值。

12.2.2 准备应用程序

在部署应用程序之前，还有一些基本的准备工作要做，以便为生产环境做好准备。在接下来的内容中，将更改错误的显示方式并设置数据库的生产环境连接字符串。

1．创建**Error**控制器和视图

目前，应用程序已配置为使用开发人员友好的错误页面，这些页面在出现问题时可以提供有用的信息。这不是最终用户应该看到的信息，因此将一

个名为ErrorController.cs的类文件添加到Controllers文件夹中，并用它定义代码清单12-12所示的简单控制器。

代码清单12-12　Controllers文件夹下的ErrorController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {

    public class ErrorController : Controller {

        public ViewResult Error() => View();
    }
}
```

Error控制器定义了呈现默认视图的Error操作方法。为了给Error控制器提供视图，创建Views/Error文件夹，添加一个名为Error.cshtml的Razor视图文件，在其中编写代码清单12-13所示的代码。

代码清单12-13　Views/Error文件夹下的Error.cshtml文件的内容

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/c
ss/bootstrap.min.css" />
    <title>Error</title>
</head>
<body>
    <h2 class="text-danger">Error.</h2>
    <h3 class="text-danger">An error occurred while pro
cessing your request.</h3>
</body>
</html>
```

这种错误页面是万不得已的选择，最好尽可能简单，而不要依赖共享视图、视图组件或其他功能。在本例中，禁用共享布局并定义一个简单的HTML文档，该HTML文档说明出现了错误，但并未提供错误的任何信息。

2. 定义生产数据库设置

下一步是创建一个文件，该文件将为应用程序提供生产环境中的数据库连接字符串。向SportsStore项目添加一个名为appsettings.production.json的ASP.NET配置文件，并添加代码清单12-14所示的内容。

提　示

解决方案资源管理器会将此文件嵌套在文件列表中的appsettings.json中，如果稍后想再次编辑该文件，就必须先展开。

代码清单12-14　appsettings.production.json文件的内容

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=tcp:sportsstorecore2d
b.database.windows.net,1433;Initial
      Catalog=products;Persist Security Info=False;User
 ID={your_username};Password={your_
      password};MultipleActiveResultSets=True;Encrypt=T
rue;TrustServerCertificate=False;
      Connection Timeout=30;"

    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=tcp:sportsstorecore2d
b.database.windows.net,1433;Initial
      Catalog=identity;Persist Security Info=False;User
 ID={your_username};Password={your_
      password};MultipleActiveResultSets=True;Encrypt=T
rue;TrustServerCertificate=False;
      Connection Timeout=30;"
    }
  }
}
```

这个文件很难阅读，因为连接字符串不能跨多行。这个文件复制了appsettings.json文件的连接字符串部分，但使用了Microsoft Azure连接字符串（别忘了替换用户名和密码占位符）。这里还将MultipleActiveResultSets设置为True，这允许多个并

发查询，并避免在执行复杂LINQ查询时的常见错误情况。

注　意

在将用户名和密码插入连接字符串时需要删除括号字符，最终结果应该是Password=MyPassword而不是Password={MyPassword}。

3．配置应用程序

现在可以更改Startup类，这样应用程序在生产环境中就会执行不同的操作。代码清单12-15显示

了所做的更改。

代码清单12-15　在Startup.cs文件中配置应用程序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<ApplicationDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProdu
cts:ConnectionString"]));
```

```
            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<IdentityUser, Identity
Role>()
                .AddEntityFrameworkStores<AppIdentityDb
Context>()
                .AddDefaultTokenProviders();

            services.AddTransient<IProductRepository, E
FProductRepository>();
            services.AddScoped<Cart>(sp => SessionCart.
GetCart(sp));
            services.AddSingleton<IHttpContextAccessor,
 HttpContextAccessor>();
            services.AddTransient<IOrderRepository, EFO
rderRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
            } else {
                app.UseExceptionHandler("/Error");
            }

            app.UseStaticFiles();
```

```
            app.UseSession();
            app.UseAuthentication();
            app.UseMvc(routes => {
                routes.MapRoute(name: "Error", template
: "Error",
                    defaults: new { controller = "Error
", action = "Error" });
                routes.MapRoute(name: null,
                    template: "{category}/Page{productP
age:int}",
                    defaults: new { controller = "Produ
ct", action = "List" }
                );
                routes.MapRoute(name: null,template: "P
age{productPage:int}",
                    defaults: new { controller = "Produ
ct",
                    action = "List", productPage = 1 }
                );
                routes.MapRoute(name: null, template: "
{category}",
                    defaults: new { controller = "Produ
ct",
                        action = "List", productPage =
1 }
                );
                routes.MapRoute(name: null,template: ""
,
                    defaults: new { controller = "Produ
ct",
                        action = "List", productPage =
1 });
                routes.MapRoute(name: null, template: "
{controller}/{action}/{id?}");
            });
            //SeedData.EnsurePopulated(app);
```

```
            //IdentitySeedData.EnsurePopulated(app);
        }
    }
}
```

IHostingEnvironment接口用于提供有关运行应用程序的环境信息，例如，当前运行环境为开发环境还是生产环境。当托管环境设置为Production时，ASP.NET Core将加载appsettings.production.json文件及其内容，用来覆盖appsettings.json文件中的设置，这意味着Entity Framework Core将连接到Microsoft Azure数据库而不是LocalDB。可以使用很多选项在不同环境中定制应用程序的配置，这将在第14章中解释。

这里还注释了数据库的填充语句，12.2.4节将解释这些语句。

12.2.3　应用数据库迁移

要使用应用程序所需的架构设置数据库，请打开新的命令提示符或PowerShell窗口，然后导航到SportsStore项目文件夹。首先需要设置环境变量，以便dotnet命令行工具使用Microsoft Azure连接字符串。如果使用的是PowerShell，请使用以下命令设置环境变量：

```
$env:ASPNETCORE_ENVIRONMENT="Production"
```

如果使用的是命令提示符，请使用以下命令设置环境变量：

```
set ASPNETCORE_ENVIRONMENT=Production
```

在SportsStore项目文件夹中运行以下命令，以将项目中的迁移应用于Microsoft Azure数据库：

```
dotnet ef database update --context ApplicationDbContex
t
dotnet ef database update --context AppIdentityDbContext
```

环境变量指定了用于获取连接字符串以访问数据库的宿主环境。如果这些命令不起作用，请确保已将Microsoft Azure防火墙配置为允许开发计算机访问（如本章前面所述），并且已正确复制和修改了连接字符串。

12.2.4 管理数据库填充

代码清单12-15注释掉了Startup类中用于填充数据库的语句。这样做是因为用于将迁移应用于数据库的Entity Framework Core命令依赖于Startup类设置的服务，这意味着在启用这些语句的情况下，将在应用迁移之前调用为数据库设定填充数据的代码，这会导致错误并阻止迁移工作。设置数据库时，这不会导致问题。但对于生产数据库，因为SeedData.EnsurePopulated方法在填充数据之前应用迁移，而且还因为在迁移应用到数据库之前，还没有将Identity种子数据添加到应用程序中，所以会出

错。

对于生产环境，建议采用不同的种子数据填充方法。对于用户账户，建议在登录时使用管理员账户填充数据库。这里将为管理工具添加一个功能，用于填充products数据库，以便生产系统可以填充测试数据，或者根据需要留空以获取实际数据。

注　意

在生产系统中填充验证数据时应谨慎处理，并且应用程序应使用第28～30章中描述的功能，在部署后立即更改密码。

1．填充身份数据

更改用户数据填充方式的第一步是简化IdentitySeedData类中的代码，如代码清单12-16所示。

代码清单12-16　简化Models文件夹下的IdentitySeedData.cs文件中的代码

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.Tasks;

namespace SportsStore.Models {

    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123
$";

        public static async Task EnsurePopulated(UserMa
nager<IdentityUser>
                userManager) {

            IdentityUser user = await userManager.FindB
yIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
```

```
                await userManager.CreateAsync(user, adm
inPassword);
            }
        }
    }
}
```

EnsurePopulated方法接收一个对象作为参数，而不是在内部获取UserManager <IdentityUser>服务。这样就可以将数据库填充集成到AccountController类中，如代码清单12-17所示。

代码清单12-17　在Controllers文件夹下的AccountController.cs文件中填充数据

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;
using SportsStore.Models;

namespace SportsStore.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManag
er;
```

```
        public AccountController(UserManager<IdentityUs
er> userMgr,
                SignInManager<IdentityUser> signInMgr)
{
            userManager = userMgr;
            signInManager = signInMgr;
            IdentitySeedData.EnsurePopulated(userMgr).W
ait();
        }

        // ...other methods omitted for brevity...
    }
}
```

这些更改将确保每次创建AccountController对象以处理HTTP请求时，都会为Identity数据库填充数据。当然，这并不是很理想，但是并没有很好的方法来为数据库填充数据，这种方法将确保应用程序可以在生产环境和开发环境中进行管理，尽管代价是产生一些额外的数据库查询。

2．填充产品数据

对于产品数据，这里将向管理员提供一个按

钮，该按钮将在数据库为空时为其提供初始数据。第一步是更改数据填充代码，以便使用一个接口，允许访问通过控制器而不是Startup类提供的服务，如代码清单12-18所示。这里还注释了自动应用挂起迁移的任何语句，这可能导致数据丢失，因此在生产环境中只能非常谨慎地使用。

代码清单12-18　准备在Models文件夹下的SeedData.cs文件中手动填充数据

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using System;

namespace SportsStore.Models {

    public static class SeedData {

        public static void EnsurePopulated(IServiceProv
ider services) {
            ApplicationDbContext context =
                services.GetRequiredService<Application
DbContext>();
            //context.Database.Migrate();
            if (!context.Products.Any()) {
```

```
                context.Products.AddRange(

                    // ...statements omiited for brevit
y...

                );
                context.SaveChanges();
            }
        }
    }
}
```

下一步是更新Admin控制器以添加将触发数据填充操作的操作方法，如代码清单12-19所示。

代码清单12-19　在Controllers文件夹下的AdminController.cs文件中填充数据库

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
 {
```

```
            repository = repo;
        }

        public ViewResult Index() => View(repository.Pr
oducts);

        // ...other methods omitted for brevity...

        [HttpPost]
        public IActionResult SeedDatabase() {
            SeedData.EnsurePopulated(HttpContext.Reques
tServices);
            return RedirectToAction(nameof(Index));
        }
    }
}
```

对新的操作方法使用HttpPost特性进行修饰，以便可以使用POST请求进行定位，并且一旦数据库被填充，就将浏览器重定向到Index操作方法。剩下的就是创建一个按钮来填充数据库，如代码清单12-20所示。

代码清单12-20　在Views/Admin文件夹下的Index.cshtml文件中添加一个按钮

```
@model IEnumerable<Product>

```

```
@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}

@if (Model.Count() == 0) {
    <div class="text-center m-2">
        <form asp-action="SeedDatabase" method="post">
            <button type="submit" class="btn btn-danger
">Seed Database</button>
        </form>
    </div>
} else {
    <table class="table table-striped table-bordered ta
ble-sm">
        <tr>
            <th class="text-right">ID</th>
            <th>Name</th>
            <th class="text-right">Price</th>
            <th class="text-center">Actions</th>
        </tr>
        @foreach (var item in Model) {
            <tr>
                <td class="text-right">@item.ProductID<
/td>
                <td>@item.Name</td>
                <td class="text-right">@item.Price.ToSt
ring("c")</td>
                <td class="text-center">
                    <form asp-action="Delete" method="p
ost">
                        <a asp-action="Edit" class="btn
 btn-sm btn-warning"
                           asp-route-productId="@item.P
roductID">
                            Edit
```

```
                        </a>
                        <input type="hidden" name="Prod
uctID"
                            value="@item.ProductID" />
                        <button type="submit" class="bt
n btn-danger btn-sm">
                            Delete
                        </button>
                    </form>
                </td>
            </tr>
        }
    </table>
}
<div class="text-center">
    <a asp-action="Create" class="btn btn-primary">Add
Product</a>
</div>
```

12.2.5　部署应用程序

要部署应用程序，请在Solution Explorer窗格中右击SportsStore项目，然后从弹出的菜单中选择Publish。Visual Studio将为你提供一系列发布方法，如图12-2所示。

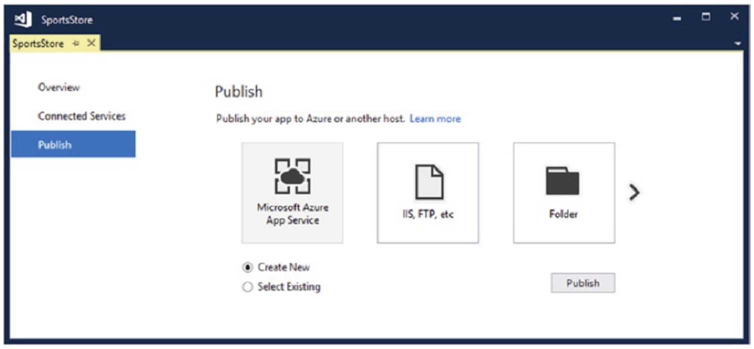


图12-2　发布方法

如果部署失败，如何处理

部署失败的最可能原因是连接字符串错误，要么因为没有从Microsoft Azure中正确复制，要么因为编辑错误，没有正确插入用户名和密码。如果部署失败，首先应该检查连接字符串。如果在“应用数据库迁移”部分没有从dotnet ef database update命令获得预期结果，部署将失败。如果命令确实有效但部署失败，那么请确保已设置环境变量，因为你可能正在使用本地数据库而不是云中的数据库。

选择Microsoft Azure App Service选项，并确保选中Create New（Select Existing用于更新现有的已部署的应用程序）。系统将提示你提供部署的详细信息。首先单击Add an Account，然后输入Microsoft Azure凭据。

输入凭据后，可以选择已部署应用程序的名称并输入服务的详细信息，具体取决于拥有的Microsoft Azure账户类型、要部署的区域以及所需的部署服务，如图12-3所示。

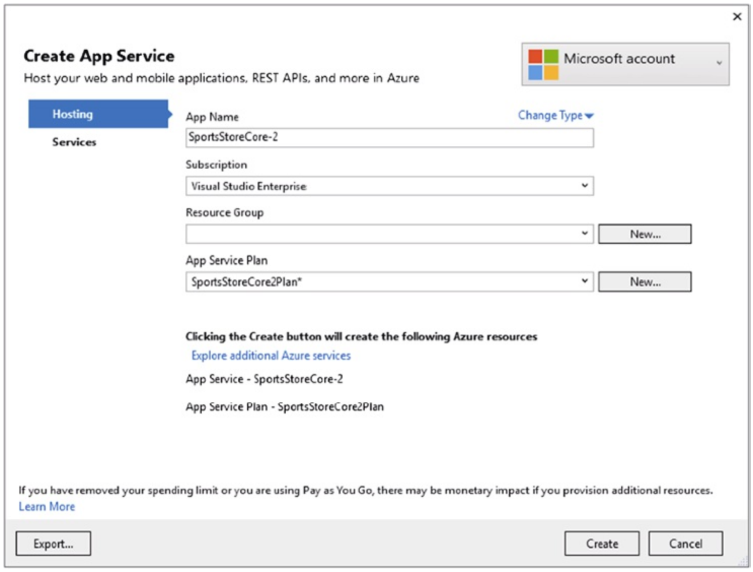


图12-3　创建新的Microsoft Azure App服务

配置完服务后，单击Create按钮。设置服务后，系统将显示发布摘要，将应用程序发布到托管服务，如图12-4所示。

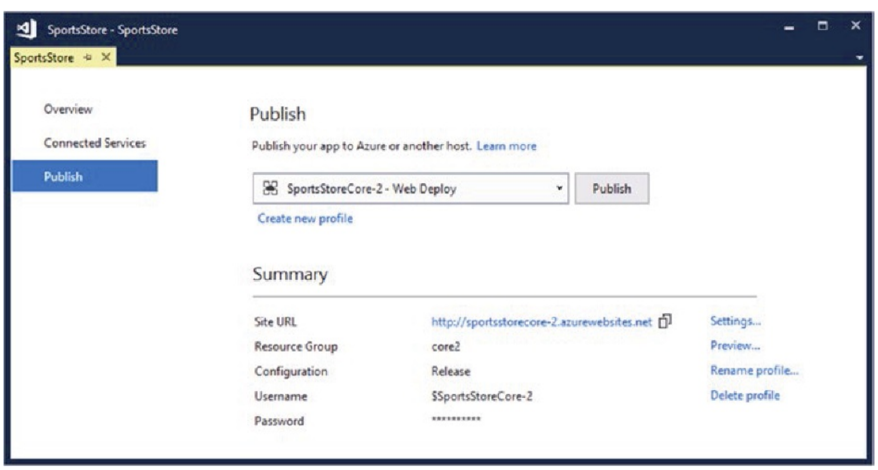


图12-4　服务的发布摘要

单击Publish按钮开始部署过程。通过从Visual Studio的菜单栏中选择View→Other Windows→Web Publish Activity，可以查看发布进度。在此过程中请耐心等待，因为将项目中的所有文件发送到Azure服务可能需要一段时间。后续更新将会更快，因为只会传输修改的文件。

部署完成后，Visual Studio将为部署好的应用程序打开一个新的浏览器窗口。由于products数据库为空，你将看到图12-5所示的布局。

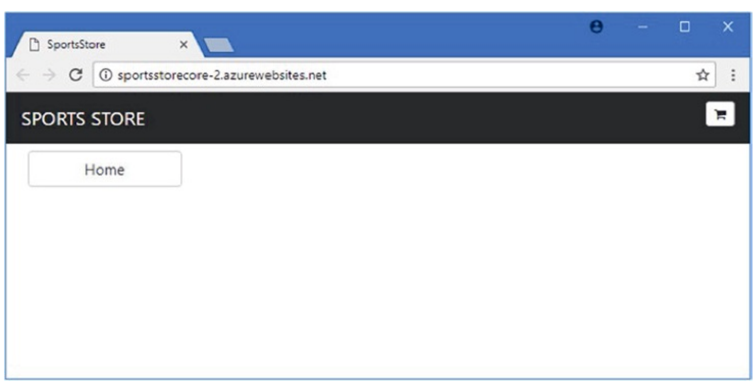


图12-5　已部署应用程序的初始状态

导航到/Admin/Index URL并使用用户名Admin

和密码Secret123$进行身份验证。Identity数据库将按需填充数据，允许你登录应用程序的管理后台，如图12-6所示。

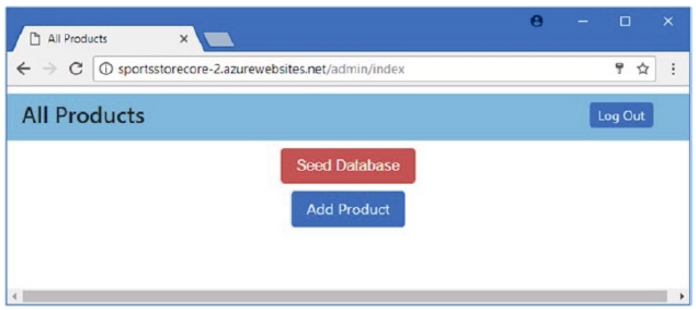


图12-6　管理页面

单击Seed Database按钮以填充products数据库，这将产生图12-7所示的结果。然后，可以导航回应用程序的根URL并正常使用。

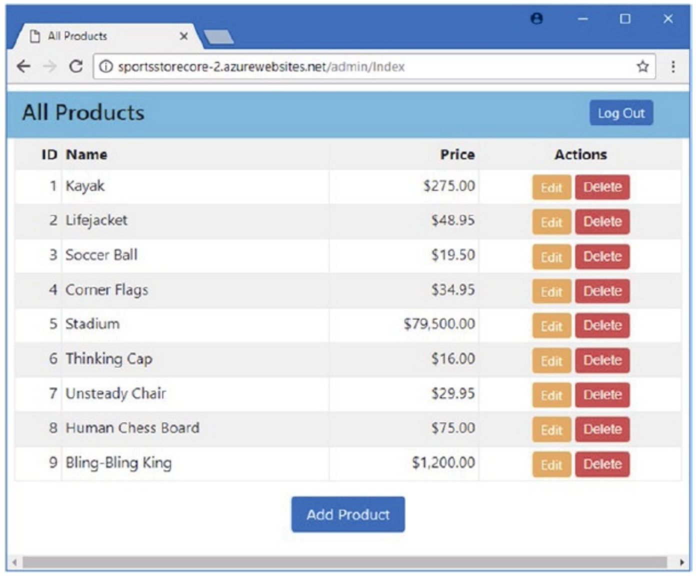


图12-7　填充数据后的数据库

12.3　小结

本章和前几章演示了如何通过ASP.NET Core MVC创建一个现实的电子商务应用程序。这个扩展的示例引入了许多关键的MVC特性——控制器、操作方法、路由、视图、元数据、验证、布局、身份验证等。本章还讲述了如何使用与MVC相关的一些关键技术，包括Entity Framework Core、依赖注入和单元测试。这个最终的SportsStore应用程序拥有

干净的、面向组件的体系结构，从而将各种关注点和易于扩展及维护的代码库分离开来。下一章将展示如何使用Visual Studio代码来创建ASP.NET Core MVC应用程序。

第13章　使用Visual Studio Code

本章将展示如何使用Visual Studio Code创建ASP.NET Core MVC应用程序，Visual Studio Code是一个由微软开发的开源跨平台编辑器。虽然名称如此，但Visual Studio Code与Visual Studio无关，Visual Studio Code基于Electron框架，很多其他Web应用程序框架（如Angular）的开发人员使用的Atom编辑器也基于这个框架。

Visual Studio Code支持Windows、macOS和最流行的Linux发行版。Visual Studio Code现已发展为强大且功能齐全的开发环境，尽管尚未提供Visual Studio的所有功能。作者自己也越来越多地使用Visual Studio Code，因为它易于使用、速度很快，而且对其他语言（如JavaScript和TypeScript）的支

持也很好。

13.1 设置开发环境

设置Visual Studio Code需要执行一些额外步骤，因为有些功能在Visual Studio中是默认提供的，但对于Visual Studio Code来说，需要由外部工具处理。其中一些工具与Visual Studio使用的工具相同，但还有一些工具对.NET开发世界来说是全新的，你可能还不熟悉。但好消息是，这些工具在其他Web应用程序框架的开发人员中已经得到广泛使用，质量和功能都很优秀。接下来将引导你安装Visual Studio Code以及MVC开发所需的基本工具和附件。

13.1.1 安装Node.js

在客户端开发中，Node.js（也称为Node）已经

成为许多流行的开发工具所依赖的运行时。Node.js是在2009年创建的，是用JavaScript为服务器端应用程序编写的简单高效的运行时。Node.js基于Chrome浏览器中使用的JavaScript引擎，并提供了在浏览器环境之外执行JavaScript代码的API。

Node.js作为应用程序服务器已经取得了一些成功，对于本章来说，Node.js为新一代跨平台的构建工具和包管理器提供了基础。Node.js团队的一些明智的设计决策和Chrome JavaScript运行时提供的跨平台支持为那些热情的工具开发者，特别是那些想要支持Web应用程序开发的人提供了机会。

注　意

有两个可以使用的Node.js版本：长期支持（LTS）版本为生产环境中的部署提供了稳定的基础，在生产环境中要尽量减少变更，LTS版本更新每6个月发布一次，并会维护18个月；当前版本是快速更新版本，注重新功能的实现，而不是稳定性。本章使用当前版本。

1．在Windows系统上安装Node.js

从Node.js网站下载并运行为Windows提供的Node.js安装程序。安装Node.js时，请确保将其添加到PATH系统变量中。图13-1显示了Windows安装程序，它提供了一个安装选项，可以将Node.js添加到PATH系统变量。

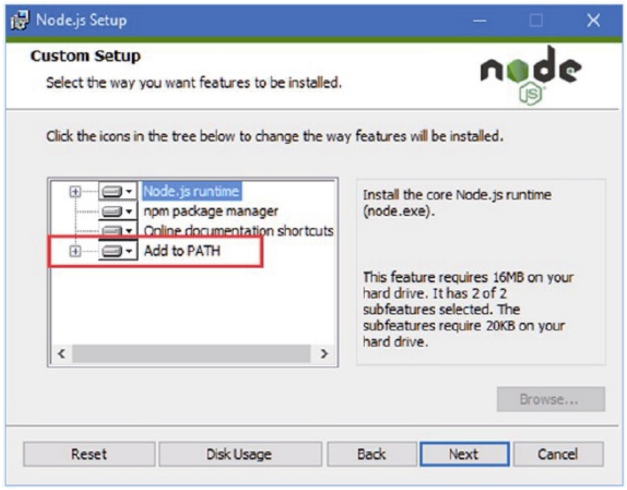


图13-1　将Node.js添加到PATH系统变量中

2．在macOS上安装Node.js

可以从Node.js网站下载用于macOS的Node.js安装程序。运行安装程序并接受默认值。安装完成后，确保/usr/local/bin位于$PATH中。

3．在Linux系统上安装Node.js

在Linux系统上安装Node.js的最简单方法是使用包管理器，Node.js团队在Node.js网站上为主要发行版提供了说明。对于Ubuntu系统，可使用代码清

单13-1中的命令下载并安装Node.js。

代码清单13-1　安装Node.js

```
sudo curl -sL https***nodesource***/setup_6.x | sudo -E
 bash -
sudo apt-get install -y nodejs
```

13.1.2　检查Node.js安装状态

安装完成后，打开新的命令提示符并运行代码清单13-2所示的命令，检查Node.js是否正常工作，并显示已安装的版本。

代码清单13-2　检查Node.js安装状态

```
node -v
```

如果安装成功并且Node.js已添加到PATH系统变量中，你将看到版本号。在撰写本书时，Node.js的当前版本是6.11.2。在根据本章中的示例执行操作时如果遇到问题，可尝试使用这个特定版本。

13.1.3　安装Git

Visual Studio Code已经集成了Git支持，但还需要单独安装Git以支持Bower工具，用于管理客户端软件包。

1．在**Windows**系统或**macOS**上安装**Git**

你可从git-scm网站下载并运行安装程序。

2．在**Linux**系统上安装**Git**

大多数Linux系统中已经安装了Git。如果你仍想单独安装Git，请参阅git-scm网站上的发行版安装说明。对于Ubuntu系统，使用代码清单13-3所示的命令安装Git。

代码清单13-3　在Ubuntu系统上安装Git

```
sudo apt-get install git
```

13.1.4 检查Git安装状态

完成安装后，在新的命令提示符或Terminal中运行代码清单13-4所示的命令，检查Git是否已安装且可用。

代码清单13-4 检查Git安装状态

```
git --version
```

以上命令会输出已安装的Git版本。在撰写本书时，用于Windows系统和macOS的最新版本的Git是2.14.1，用于Linux系统的最新版本的Git是2.7.4。

13.1.5 安装Bower

Node.js附带了Node Package Manager（NPM），用于下载和安装使用JavaScript编写的软件包。本章所需的唯一软件包是Bower，它用于管理客户端软件包，第6章对此进行了说

明。运行代码清单13-5所示的命令，为Windows系统下载并安装Bower。

代码清单13-5　在Windows系统上安装Bower

```
npm install -g bower@1.8.0
```

对于Linux系统和macOS，可使用相同的命令，但需要sudo，如代码清单13-6所示。

代码清单13-6　在Linux系统或macOS上安装Bower

```
sudo npm install -g bower@1.8.0
```

13.1.6　安装.NET Core

ASP.NET Core MVC开发需要.NET Core运行时。每个受支持的平台都有自己的安装过程，参见Microsoft官网。微软提供了用于Windows系统和macOS的安装程序，并使用tar存档提供Linux系统下的安装命令。

1．在**Windows**系统和**macOS**上安装**.NET Core**

要在Windows系统或macOS上安装.NET Core，只需要下载并运行.NET Core SDK安装程序即可。

2．在**Linux**系统上安装**.NET Core**

微软在其官网上提供了在最流行的Linux发行版上安装.NET Core的说明。本章使用的是Ubuntu，首先使用代码清单13-7所示的命令为apt-get设置新的种子。

代码清单13-7　准备在Ubuntu Linux系统上安装.NET Core

```
sudo sh -c 'echo "deb [arch=amd64] *****://apt-mo.traff
icmanager.***/repos/dotnet-release/
xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver ***://keyserver.ubuntu.***
:80 --recv-keys 417A0893
sudo apt-get update
```

下一步是安装.NET Core，如代码清单13-8所示。

代码清单13-8　在Ubuntu Linux系统上安装.NET Core

```
sudo apt-get install dotnet-sdk-2.0.0
```

13.1.7　检查.NET Core安装状态

无论使用何种平台，都可以检查.NET Core是否已安装并可以使用。打开新的命令提示符或Terminal并运行代码清单13-9所示的命令。

代码清单13-9　检查.NET Core版本

```
dotnet --version
```

dotnet命令会启动.NET运行时，并显示已安装的.NET Core的版本号。在撰写本书时，当前版本是2.0.0，但是到你阅读本书时，可能已经有更新的版本了。

13.1.8　安装Visual Studio Code

最重要的一步是下载并安装Visual Studio Code，安装包可从Visual Studio官网获得。安装包可用于Windows系统、macOS和流行的Linux系统发行版。下载并安装所选平台的安装包。

注　意

微软每个月都会发布新版的Visual Studio Code，这意味着你安装的版本可能会与本书的版本不同。虽然基本原理保持不变，但这意味着可能需要进行一些尝试才能完成本章中的一些示例。

1．在Windows系统上安装Visual Studio Code

要在Windows系统上安装Visual Studio Code，只需要运行安装程序即可。完成安装过程后，Visual Studio Code将启动，你将看到编辑器窗口，如图13-2所示。

2．在**macOS**上安装**Visual Studio Code**

Visual Studio Code可作为Mac计算机的zip存档提供，可以从Microsoft官网下载。展开存档，然后双击其中包含的Code.app文件用于启动Visual Studio Code，生成图13-2所示的编辑器窗口。

3．在**Linux**系统上安装**Visual Studio Code**

微软为Debian和Ubuntu系统提供了.deb文件，为Red Hat系统、Fedora系统和CentOS提供了.rpm文件。下载并安装适用于对应Linux发行版的文件。由于本章使用的是Ubuntu系统，因此下载.deb文件

并使用Ubuntu软件工具进行安装。

安装完成后，运行代码清单13-10所示的命令以启动Visual Studio Code，生成图13-2所示的编辑器窗口。

代码清单13-10　在Linux系统上启动Visual Studio Code

```
/usr/share/code/code
```

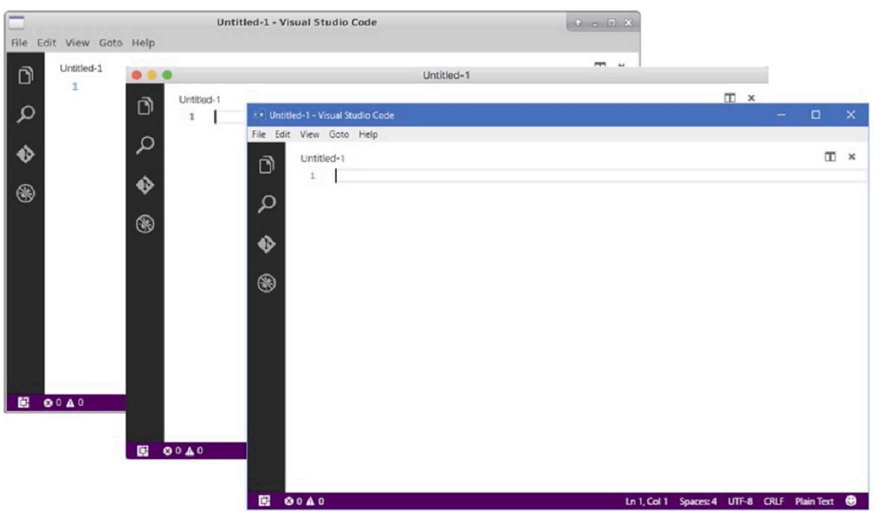


图13-2　在Windows系统、macOS和Ubuntu Linux系统上运行Visual Studio Code

13.1.9　检查Visual Studio Code安装状态

成功安装Visual Studio Code后，就可以启动应用程序并查看编辑器了（这里更改了配色方案，因为默认的深色主题不适合图书出版）。

13.1.10　安装Visual Studio Code的C#扩展

Visual Studio Code通过扩展来支持特定于语言的功能，但这些扩展与Visual Studio支持的扩展不同。对ASP.NET Core MVC开发而言最重要的扩展是增加对C#的支持，这看起来像是粗心的遗漏，但反映了如下事实：微软已将Visual Studio Code定位为支持尽可能广泛的语言和框架的通用跨平台编辑器。

要安装C#扩展，请单击Visual Studio Code窗口左侧的Extensions图标。在搜索框中输入csharp，在列表中找到C# for Visual Studio Code扩展，如图13-3所示。

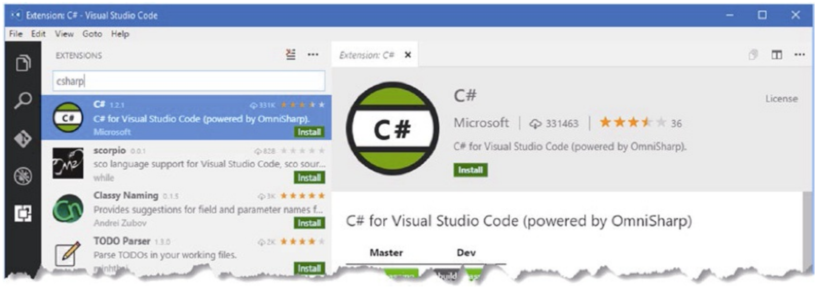


图13-3　找到C# for Visual Studio Code扩展

单击Install按钮，Visual Studio Code将下载并安装扩展。单击Enable按钮重新启动Visual Studio Code并启用for Visual Studio Code扩展，如图13-4所示。

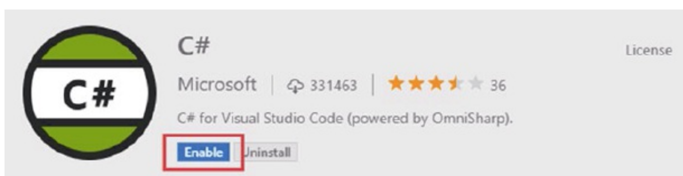


图13-4　启用C#扩展

13.2　创建ASP.NET Code项目

Visual Studio Code没有集成创建ASP.NET Core项目的功能，但可以使用dotnet命令行创建新项

目。

创建正确的文件夹结构以正确设置项目非常重要，尤其是在使用单元测试时。在合适的位置创建名为InvitesProjects的项目。

接下来，在InvitesProjects文件夹中创建一个名为PartyInvites的文件夹，使用命令提示符导航到该文件夹并运行代码清单13-11所示的命令。

代码清单13-11　在PartyInvites文件夹中创建一个新的项目

```
dotnet new web --language C# --framework netcoreapp2.0
```

dotnet new命令能够以命令行的方式访问项目模板，代码清单13-11指定的Web模板对应于前面章节中使用的Visual Studio Empty模板。表13-1描述了可用于ASP.NET Core开发的一组项目模板（其他模板也是可用的，但它们不适用于ASP.NET Core。运

行dotnet new --help命令可查看完整列表）。

表**13-1**　用于**ASP.NET Core**开发的**dotnet new**模板

名称	描述
web	这是前面章节中使用的Empty模板，用于创建ASP.NET Core项目，但是没有启用MVC框架
mvc	这是第2章中使用的Web Application(Model-View-Controller)模板，用于创建包含MVC框架、占位符控制器和视图的ASP.NET Core项目
xunit	这是xUnit Test Project（.NET Core）模板，可使用xUnit包设置单元测试

13.3　使用Visual Studio Code准备项目

要在Visual Studio Code中打开项目，请从File菜单中选择Open Folder，导航到InvitesProjects项目文件夹，然后单击Select Folder按钮。Visual Studio Code将打开项目并自动安装编辑和调试C#应用程序所需的一些软件包。第一次开始编辑文件几秒后，你将看到一条消息，提示将一些资源添加到项目

中，如图13-5所示。



图13-5　提示将资源添加到项目中

单击Yes按钮，Visual Studio Code将创建.vscode文件夹并添加一些用于配置构建过程的文件。默认情况下，Visual Studio Code使用包含3个区域的布局。侧栏（在图13-6中已框选）提供对主要功能区域的访问。顶层的按钮用于打开资源管理器窗格，显示已打开文件夹的内容。其他按钮提供对搜索功能、集成源代码管理、调试器和已安装扩展集的访问。

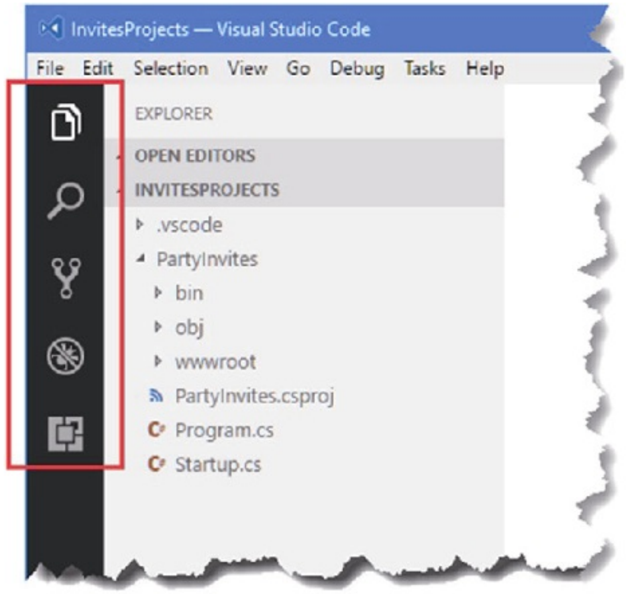


图13-6　Visual Studio Code的边栏

单击资源管理器窗格中的文件，即可打开并进行编辑。Visual Studio Code支持同时编辑多个文件，可以通过单击窗口右上角的Split Editor按钮来创建新的编辑器窗格。Visual Studio代码编辑器相当不错，在输入NuGet和Bower软件包的名称和版本时，可以提供良好的智能感知支持。

除项目文件夹中的内容之外，资源管理器窗格还会显示当前正在编辑的文件，这使你可以轻松地

将注意力集中在正在使用的文件集上，这在处理大型项目中的相关文件子集时非常有用。

13.3.1　管理客户端软件包

Bower用于管理Visual Studio Code项目中的客户端软件包，就像在Visual Studio中一样，但是还需要做一些额外的工作。

第一步是添加一个名为.bowerrc的文件，该文件用于告诉Bower在哪里安装软件包。右击PartyInvites文件夹，从弹出菜单中选择New File，创建新文件，如图13-7所示。

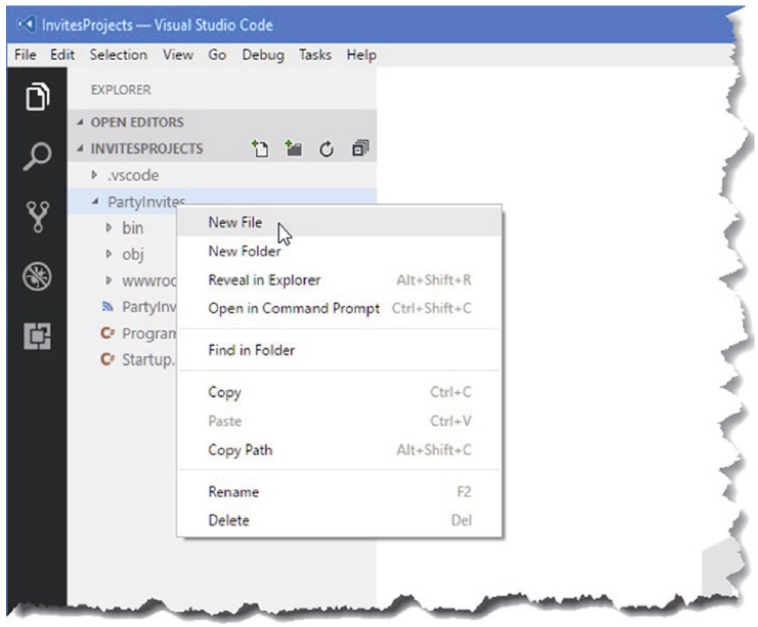


图13-7　创建新文件

将文件名设置为.bowerrc（请注意，文件名中有两个r）并添加代码清单13-12所示的内容。

代码清单13-12　.bowerrc文件的内容

```
{
  "directory": "wwwroot/lib"
}
```

接下来，创建一个名为bower.json的文件，并添加代码清单13-13所示的内容。

代码清单13-13　bower.json文件的内容

```
{
  "name": "PartyInvites",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

使用命令提示符或Terminal在PartyInvites文件夹中运行代码清单13-14所示的命令，从而使用Bower工具下载并安装bower.json文件中指定的客户端软件包。

代码清单13-14　安装客户端软件包

```
bower install
```

13.3.2　配置应用程序

项目的初始化过程会创建一个不支持MVC的空项目。代码清单13-15显示了如何使用最基本配

置在Startup.cs文件中添加对MVC的支持，其中的语句将在第14章中进行描述。

代码清单13-15　在Startup.cs文件中添加对MVC的支持

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace PartyInvites {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

13.3.3　构建和运行项目

要构建和运行项目，请在PartyInvites目录中运行代码清单13-16所示的命令。

代码清单13-16　运行应用程序

```
dotnet run
```

Visual Studio Code将编译项目中的代码，并使用第14章描述的Kestrel应用程序服务器来运行应用程序，等待端口5000上的HTTP请求。

Visual Studio Code不支持检测C#类文件发生的更改，因此在进行更改时必须停止应用程序并再次启动。

要测试应用程序，请使用浏览器导航到http://localhost：5000。你将看到图13-8所示的响应，显示404错误是因为项目中没有控制器用来处

理请求。

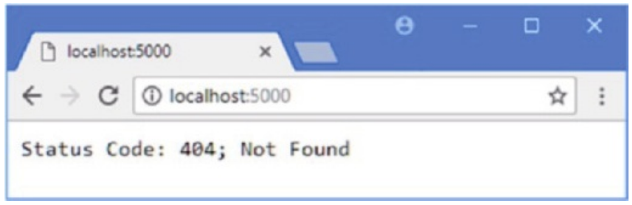


图13-8　测试示例应用程序

13.4　重新创建PartyInvites应用程序

所有准备工作都已完成，这意味着可以将注意力转移到创建MVC应用程序。这里将从第2章开始重新创建简单的PartyInvites应用程序，但需要进行一些更改以重点展示如何使用Visual Studio Code。

13.4.1　创建模型和存储库

请右击PartyInvites文件夹，然后从弹出菜单中选择New Folder，创建新文件夹，如图13-9所示。将文件夹的名称设置为Models。

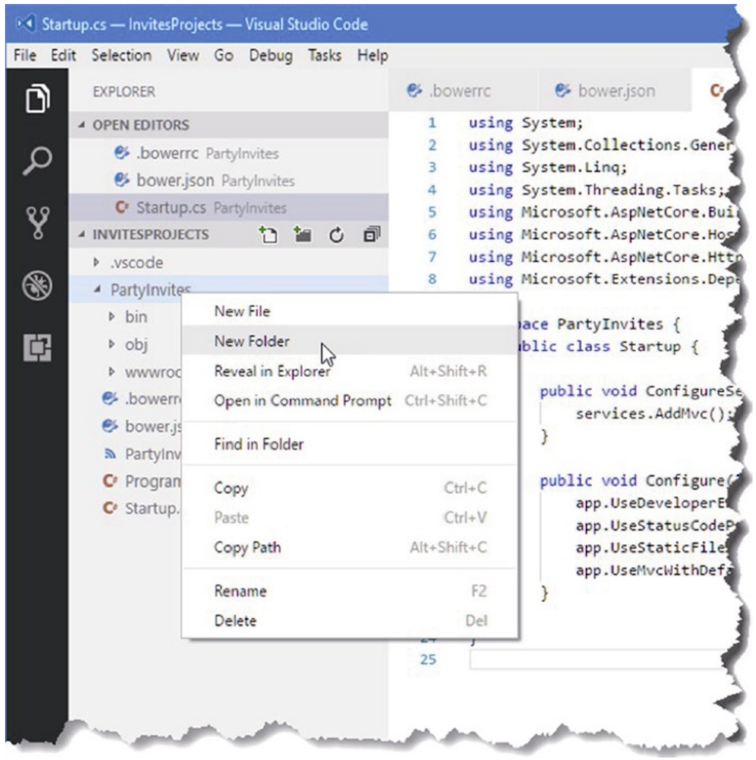


图13-9　创建新文件夹

右击资源管理器中的Models文件夹，从弹出菜单中选择New File，将文件名设置为GuestResponse.cs，然后添加代码清单13-17所示的C#代码。

使用Visual Studio Code编辑器

Visual Studio Code（以及本章前面安装的C#扩展）为C#文件以及JavaScript、CSS和普通HTML文件等常见Web格式提供了完整的编辑体验。在Visual Studio Code中编写MVC应用程序与使用Visual Studio编辑器有很多共同点，如智能感知支持、代码着色和高亮显示错误（以及提供修复建议）等。

Visual Studio Code的主要缺点是缺乏自定义功能，特别是在格式化代码时。在写作本书时，还有其他语言可用的配置选项，但C#扩展没有提供自定义功能。如果你喜欢的编码规范不是默认支持的，那么C#扩展可能有点难以使用。但总的来说，编辑器响应迅速且易于使用，在macOS或Linux系统上编写MVC应用程序的体验并不差。

代码清单13-17　PartyInvites/Models文件夹下的GuestResponse.cs文件的内容

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

    public class GuestResponse {

        public int id {get; set; }

        [Required(ErrorMessage = "Please enter your nam
e")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter your ema
il address")]
        [RegularExpression(".+\\@.+\\..+",
            ErrorMessage = "Please enter a valid email
address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your pho
ne number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whethe
r you'll attend")]
        public bool? WillAttend { get; set; }
    }
}
```

接下来，将名为IRepository.cs的文件添加到Models文件夹中，并用它定义代码清单13-18所示的接口。本章中的应用程序与第2章中的应用程序之间最重要的区别是将模型数据存储在持久化数据库中。IRepository接口描述了应用程序如何访问模型数据而不指定实现。

代码清单13-18　PartyInvites/Models文件夹下的IRepository.cs文件的内容

```
using System.Collections.Generic;

namespace PartyInvites.Models {

    public interface IRepository {
        IEnumerable<GuestResponse> Responses {get; }

        void AddResponse(GuestResponse response);
    }
}
```

将一个名为ApplicationDbContext.cs的文件添加到Models文件夹中，并用它定义数据库上下文类，

如代码清单13-19所示。

代码清单13-19　PartyInvites/Models文件夹下的ApplicationDbContext.cs文件的内容

```
using Microsoft.EntityFrameworkCore;

namespace PartyInvites.Models {
    public class ApplicationDbContext : DbContext {
        public ApplicationDbContext() {}

        protected override void OnConfiguring(DbContext
OptionsBuilder builder) {
            builder.UseSqlite("Filename=./PartyInvites.
db");
        }

        public DbSet<GuestResponse> Invites {get; set;}
    }
}
```

SQLite将数据存储在由上下文类指定的文件中。对于示例应用程序，数据将存储在名为PartyInvites.db的文件中，该文件定义在OnConfiguring方法中。

要完成存储和访问模型数据所需的类，需要实现数据库上下文类的IRepository接口。将名为EFRepository.cs的新文件添加到Models文件夹中，并添加代码清单13-20所示的代码。

代码清单13-20　PartyInvites/Models文件夹下的EFRepository.cs文件的内容

```
using System.Collections.Generic;

namespace PartyInvites.Models {
    public class EFRepository : IRepository {
        private ApplicationDbContext context = new Appl
icationDbContext();

        public IEnumerable<GuestResponse> Responses =>
context.Invites;

        public void AddResponse(GuestResponse response)
 {
            context.Invites.Add(response);
            context.SaveChanges();
        }
    }
}
```

EFRepository类使用与第8章中对应的类似模式

来设置SportsStore数据库。代码清单13-21在Startup类的ConfigureServices方法中添加了一条配置语句，该方法告诉ASP.NET在依赖注入功能要求实现IRepository接口时创建EFRepository类（参考第18章）。

代码清单13-21　在PartyInvites文件夹下的Startup.cs文件中配置存储库

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using PartyInvites.Models;

namespace PartyInvites {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IRepository, EFReposi
tory>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

13.4.2 创建数据库

在本书的其他部分，每当需要演示数据持久化功能时，都会使用LocalDB，LocalDB是Microsoft SQL Server的简化版本。但是LocalDB仅在Windows平台上可用，这意味着在其他平台上创建ASP.NET Core MVC应用程序时需要替代方案。LocalDB的最佳替代方案是SQLite，SQLite是跨平台的并且无须配置的数据库，可以嵌入应用程序中。在通常与ASP.NET Core MVC应用程序一起使用的数据访问层Entity Framework Core中，微软已默认包含对SQLite的支持。接下来介绍将SQLite添加到项目中

并将其作为数据存储的过程。

使用SQLite进行开发

LocalDB是一个非常有用的工具，原因之一是它允许使用SQL Server数据库引擎进行开发，这使得向SQL Server生产环境的过渡变得非常简单而且基本没有风险。SQLite也是优秀的数据库，但它不适合大规模的Web应用程序，这意味着在部署MVC应用程序时需要转换到另一个数据库。使用第14章描述的项目配置功能可以简化配置更改，但是需要在暂存环境中彻底测试应用程序，以显示生产数据库引入的任何差异。

如果不确定是否在生产环境中使用SQLite，请参阅SQLite官网，上面提供了有关SQLite的优势及缺陷的介绍。

需要注意的一个问题是，Entity Framework Core可以为其他数据库生成完整的架构更改，但SQLite并不支持这个功能。在开发中使用SQLite时，这通常不是问题，因为可以删除数据库文件并重新生成具有干净架构的新文件。但是，如果考虑使用SQLite部署应用程序，则确实会使问题复杂化。

如果要在开发和生产中使用相同的数据库，请参阅readthedocs网站上受Entity Framework Core支持的数据库列表。在写作本书时这个列表还很短，但微软已经宣布支持比SQLite更适合部署的数据库，而且也可以在非Windows平台上运行。

---

1. 添加数据库包

必须手动将包含创建和应用数据库迁移命令行工具的NuGet包添加到项目中。打开PartyInvites.csproj文件并添加代码清单13-22所示的元素。

代码清单13-22　将NuGet包添加到PartyInvites文件夹下的PartyInvites.csproj文件中

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFr
ameworkCore.Tools.DotNet"
        Version="2.0.0" />
  </ItemGroup>

</Project>
```

保存更改，并在PartyInvites文件夹中运行代码清单13-23所示的命令，以确保下载并安装新的软件包。

代码清单13-23　安装NuGet包

```
dotnet restore
```

2．创建和应用数据库迁移

创建数据库与Visual Studio使用的过程相同。要创建数据库迁移，请在PartyInvites文件夹中运行代码清单13-24所示的命令。

代码清单13-24　创建数据库迁移

```
dotnet ef migrations add Initial
```

Entity Framework Core将创建一个名为Migrations的文件夹，其中包含将用于设置数据库架构的C#类。要应用数据库迁移，请在PartyInvites

文件夹中运行代码清单13-25所示的命令，从而在PartyInvites文件夹中创建数据库。

代码清单13-25　应用数据库迁移

```
dotnet ef database update
```

Visual Studio Code不支持查看SQLite数据库，但可以在sqlitebrowser网站上找到适用于Windows系统、macOS和Linux系统的优秀开源工具。

13.4.3　创建控制器和视图

在本节中，将控制器和视图添加到应用程序中。首先创建PartyInvites/Controllers文件夹并添加一个名为HomeController.cs的文件，用来创建代码清单13-26所示的控制器。

代码清单13-26　PartyInvites/Controllers文件夹下的HomeController.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) =>
            this.repository = repo;

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Mornin
g" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public ViewResult RsvpForm() => View();

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestR
esponse) {
            if (ModelState.IsValid) {
                repository.AddResponse(guestResponse);
                return View("Thanks", guestResponse);
            } else {
                // there is a validation error
                return View();
            }
        }
        public ViewResult ListResponses() =>
```

```
            View(repository.Responses.Where(r => r.Will
Attend == true));
    }
}
```

为了设置内置的标签助手，创建PartyInvites/Views文件夹，并添加一个名为_ViewImports.cshtml的文件，其中包含代码清单13-27所示的代码。

代码清单13-27　PartyInvites/Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

接下来，创建PartyInvites/Views/Home文件夹，并添加一个名为MyView.cshtml的文件，作为代码清单13-26中Index操作方法选择的视图，其中的内容如代码清单13-28所示。

代码清单13-28　PartyInvites/Views/Home文件夹下的MyView.cshtml文件的内容

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Index</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body class="p-2">
    <div class="text-center">
        <h3>We're going to have an exciting party!</h3>
        <h4>And you are invited</h4>
        <a class="btn btn-primary" asp-action="RsvpForm
">RSVP Now</a>
    </div>
</body>
</html>
```

在PartyInvites/Views/Home文件夹中添加一个名为RsvpForm.cshtml的文件，并添加代码清单13-29所示的内容。这个视图用于提供受邀者填写的HTML表单，以便他们接受或拒绝聚会邀请。

代码清单13-29　PartyInvites/Views/Home文件夹下的

RsvpForm.cshtml文件的内容

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body>
    <div class="m-2">
        <div class="text-center"><h4>RSVP</h4></div>
        <form class="p-1" asp-action="RsvpForm" method=
"post">
            <div asp-validation-summary="All"></div>
            <div class="form-group">
                <label asp-for="Name">Your name:</label
>
                <input class="form-control" asp-for="Na
me" />
            </div>
            <div class="form-group">
                <label asp-for="Email">Your email:</lab
el>
                <input class="form-control" asp-for="Em
ail" />
```

```
            </div>
            <div class="form-group">
                <label asp-for="Phone">Your phone:</lab
el>
                <input class="form-control" asp-for="Ph
one" />
            </div>
            <div class="form-group">
                <label>Will you attend?</label>
                <select class="form-control" asp-for="W
illAttend">
                    <option value="">Choose an option</
option>
                    <option value="true">Yes, I'll be t
here</option>
                    <option value="false">No, I can't c
ome</option>
                </select>
            </div>
            <div class="text-center">
                <button class="btn btn-primary" type="s
ubmit">
                    Submit RSVP
                </button>
            </div>
        </form>
    </div>
</body>
</html>
```

下一个视图文件名为Thanks.cshtml，也创建在PartyInvites/Views/Home文件夹中，内容如代码清

单13-30所示，当来宾提交响应时将显示这个视图。

代码清单13-30　PartyInvites/Views/Home文件夹下的Thanks.cshtml文件的内容

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Thanks</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
</head>
<body class="text-center">
    <p>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks
 are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but
 thanks for letting us know.
        }
```

```
    </p>
    Click <a asp-action="ListResponses">here</a>
    to see who is coming.
</body>
</html>
```

最后一个视图名为ListResponses.cshtml，与本例中的其他视图一样，也添加到PartyInvites/Views/Home文件夹中。这个视图使用代码清单13-31所示的标记显示来宾响应列表。

代码清单13-31　PartyInvites/Views/Home文件夹下的ListResponses.cshtml文件的内容

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link rel="stylesheet" href="/lib/bootstrap/dist/cs
s/bootstrap.css" />
    <title>Responses</title>
</head>
```

```
<body>
    <div class="m-1 p-1">
        <h2>Here is the list of people attending the pa
rty</h2>
        <table class="table table-sm table-striped tabl
e-bordered">
            <thead>
                <tr><th>Name</th><th>Email</th><th>Phon
e</th></tr>
            </thead>
            <tbody>
                @foreach (PartyInvites.Models.GuestResp
onse r in Model) {
                    <tr><td>@r.Name</td><td>@r.Email</t
d><td>@r.Phone</td></tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

在PartyInvites项目中运行dotnet run命令以编译项目并启动ASP.NET Core运行时。应用程序启动后，可以通过导航到http://localhost:5000来查看已完成的应用程序，如图13-10所示。

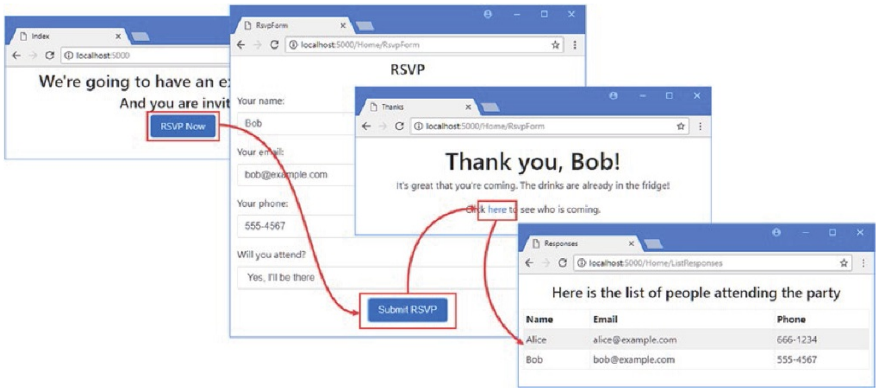


图13-10　运行完成的应用程序

13.5　Visual Studio Code中的单元测试

使用Visual Studio Code进行单元测试的过程与Visual Studio类似。第一步是为单元测试创建单独的项目。在InvitesProject文件夹中创建一个名为Tests的文件夹，并在这个文件夹中运行代码清单13-32所示的命令，以创建单元测试项目。

代码清单13-32　创建单元测试项目

```
dotnet new xunit --language C# --framework netcoreapp2.0
```

在Tests文件夹中运行代码清单13-33所示的命

令，添加对应用程序项目的引用，以引用其中包含的类用于测试。

代码清单13-33　添加对应用程序项目的引用

```
dotnet add reference ../PartyInvites/PartyInvites.csproj
```

13.5.1　创建单元测试

创建单元测试的过程如第7章所述。将一个名为HomeControllerTests.cs的类文件添加到Tests文件夹中，其中的内容如代码清单13-34所示。

代码清单13-34　Tests文件夹下的HomeControllerTests.cs文件的内容

```
using System;
using System.Collections.Generic;
using PartyInvites.Controllers;
using PartyInvites.Models;
using Xunit;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace Tests {
```

```
    public class HomeControllerTests {

        [Fact]
        public void ListActionFiltersNonAttendees() {
            //Arrange
            HomeController controller = new HomeControl
ler(new FakeRepository());
            // Act
            ViewResult result = controller.ListResponse
s();
            // Assert
            Assert.Equal(2, (result.Model as IEnumerabl
e<GuestResponse>).Count());
        }
    }

    class FakeRepository : IRepository {
        public IEnumerable<GuestResponse> Responses =>
            new List<GuestResponse> {
                new GuestResponse { Name = "Bob", WillA
ttend = true },
                new GuestResponse { Name = "Alice", Wil
lAttend = true },
                new GuestResponse { Name = "Joe", WillA
ttend = false }
            };

        public void AddResponse(GuestResponse response)
 {
            throw new NotImplementedException();
        }
    }
}
```

这是一个标准的xUnit测试，用于检查Home控制器中的ListResponses操作，并正确过滤存储库中WillAttend属性为false的GuestResponse对象。

13.5.2　运行测试

要在项目中运行单元测试，请在Tests文件夹中运行代码清单13-35所示的命令。

代码清单13-35　运行单元测试

```
dotnet test
```

这将运行项目中的所有测试并显示结果，产生如下输出：

```
Starting test execution, please wait...
[xUnit.net 00:00:00.6731479]   Discovering: Tests
[xUnit.net 00:00:00.7900132]   Discovered:  Tests
[xUnit.net 00:00:00.8432715]   Starting:    Tests
[xUnit.net 00:00:00.9967614]   Finished:    Tests

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
```

```
Test execution time: 1.6974 Seconds
```

以上结果显示有两个测试，因为项目模板包含一个名为UnitTest1.cs的文件，而其中又包含一个空的单元测试。可以删除此文件，如第7章所述。

13.6 小结

本章简要介绍了如何使用Visual Studio Code，它是一个轻量级的开发工具，支持在Windows系统、macOS和Linux系统上进行ASP.NET Core MVC开发。Visual Studio Code还无法成为完整的Visual Studio产品的替代品，但它提供了创建MVC应用程序所需的核心功能，微软每月都会发布更新以增强其功能。

本书第一部分到此结束，第二部分将深入研究ASP.NET Core的细节，并详细展示如何使用这些功能来创建应用程序。

第二部分 ASP.NET Core MVC详解

到目前为止，你已经了解了ASP.NET Core MVC存在的原因，并了解了其体系结构和底层设计目标。通过构建一个真实的电子商务应用程序，你已经掌握了测试驱动开发方式。现在是时候打开盖子并展示框架机制的全部细节了。

本书第二部分将深入讨论细节。该部分首先探讨ASP.NET Core MVC应用程序的结构以及处理请求的方式，然后重点关注各个功能，例如路由、控制器和action、MVC视图和标签助手系统，以及将MVC与域模型一起使用的方式。

第14章　配置应用程序

配置这个主题似乎并不是很有趣，但它揭示了很多MVC应用程序如何工作以及如何处理HTTP请求。你不应该跳过本章，而应该花时间了解配置系统构造MVC Web应用程序的方式，从而为理解后面的章节奠定坚实的基础。

本章将解释如何配置MVC应用程序，并展示MVC如何构建ASP.NET Core平台提供的功能。表14-1总结了在上下文中配置应用程序时的一些问题。

表**14-1**　在上下文中进行配置时的一些问题

问题	答案
配置是什么？	Program和Startup类以及JSON文件用于配置应用程序的工作方式以及依赖的包

有什么用？	配置系统允许应用程序根据运行环境进行定制，并管理依赖的
怎么用？	最重要的组件是Startup类，它用于创建服务（在整个应用程序中提供通用功能的对象）和中间件（用于处理HTTP请求）
有任何问题或限制吗？	在复杂的应用程序中，配置可能变得难以管理
还有其他选择吗？	没有。配置系统是ASP.NET和MVC应用程序设置方式的组成部分

表14-2列出了本章要介绍的操作。

表14-2　本章要介绍的操作

操作	方法	代码清单
向应用程序添加功能	将NuGet包添加到.csproj文件中	代码清单14-5～代码清单14-8
管理ASP.NET应用程序的初始化	使用Program类	代码清单14-9～代码清单14-11
配置应用程序	使用Startup类的ConfigureServices和Configure方法	代码清单14-12和代码清单14-13

创建通用功能	使用ConfigureServices方法创建服务	代码清单14-14～代码清单14-16
生成内容响应	创建内容生成中间件	代码清单14-17～代码清单14-19
阻止请求遍历请求管道	创建短路中间件	代码清单14-20和代码清单14-21
在其他中间件处理请求之前编辑请求	创建请求编辑中间件	代码清单14-22～代码清单14-24
编辑由其他中间件处理的响应	创建响应编辑中间件	代码清单14-25和代码清单14-26
设置MVC功能	使用UseMvc或UseMvcWithDefaultRoute方法	代码清单14-27
为不同环境改变应用程序配置	使用托管环境服务	代码清单14-28
处理应用程序错误	使用开发环境或生产环境错误处理中间件	代码清单14-29和代码清单14-30
在开发期间管理多个浏览器	使用浏览器链接	代码清单14-31
启用图像、JavaScript文件和CSS文件	启用静态内容中间件	代码清单14-32
从C#代码中分离配置数据	创建外部配置源，例如JSON文件	代码清单14-33～代

		码清单14-35
记录应用数据	使用日志记录中间件	代码清单14-36～代码清单14-38
准备依赖注入以与Entity Framework Core一起使用	禁用范围验证	代码清单14-39
配置MVC服务	使用选项功能	代码清单14-40
配置复杂的应用程序	使用多个外部文件或类	代码清单14-41～代码清单14-45

14.1　准备示例项目

本章中，我们将使用Empty模板创建一个名为ConfigureApps的新项目。稍后将配置应用程序，但是在做出更改之前，首先需要做一些基本的准备工作。

我们将在本章中使用Bootstrap来设置HTML内容的样式，因此使用Bower Configuration File模板创建bower.json文件，并添加代码清单14-1所示的

包。

代码清单14-1　在ConfiguringApps文件夹下的bower.json文件中添加Bootstrap

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

接下来，创建Controllers文件夹并添加一个名为HomeController.cs的类文件，用来定义代码清单14-2所示的控制器。

代码清单14-2　Controllers文件夹下的HomeController.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
```

```
        public ViewResult Index() => View(new Dictionar
y<string, string> {
                ["Message"] = "This is the Index action
"
            });
    }
}
```

接下来，创建Views/Home文件夹并添加一个名为Index.cshtml的视图文件，其中的内容如代码清单14-3所示。

代码清单14-3　Views/Home文件夹下的Index.cshtml文件的内容

```
@model Dictionary<string, string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
    <title>Result</title>
</head>
<body class="p-1">
    <table class="table table-condensed table-bordered
table-striped">
```

```
        @foreach (var kvp in Model) {
            <tr><th>@kvp.Key</th><td>@kvp.Value</td></t
r>
        }
    </table>
</body>
</html>
```

视图中的link元素依赖于内置的标签助手来选择Bootstrap CSS文件。为了启用内置的标签助手，可使用MVC View Imports Page模板来创建Views文件夹中的_ViewImports.cshtml文件，并添加代码清单14-4所示的代码。

代码清单14-4　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

启动应用程序，你将看到如图14-1所示的消息。

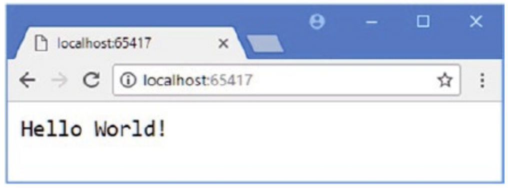


图14-1　运行示例应用程序

14.2　配置项目

最重要的配置文件是<projectname>.csproj，以取代早期版本的ASP.NET Core中使用的project.json文件。这个文件在示例项目中名为InstallingApps.csproj，Visual Studio隐藏了该文件，必须通过右击Solution Explorer窗格中的项目并从弹出的菜单中选择Edit ConfiguringApps.csproj来访问。代码清单14-5显示了ConfigureApps.csproj文件的初始内容，该文件由Visual Studio作为Empty项目模板的一部分创建。

代码清单14-5　ConfiguringApps文件夹下的ConfigureApps.csproj文件的内容

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
  </ItemGroup>

</Project>
```

.csproj文件用于配置MSBuild工具，MSBuild工具用于构建.NET项目。使用XML元素执行配置，表14-3描述了默认配置文件中的元素。尽管我们在后面的示例中使用了其他配置元素，但表14-3中的元素对于开发ASP.NET Core MVC项目已经足够了。

表**14-3**　默认**.csproj**文件中的**XML**配置元素

元素	描述

Project	此为根元素，表示这是一个MSBuild配置文件。Sdk属性被设置为Microsoft.NET.Sdk.Web，以导入构建项目所需的隐式包
PropertyGroup	此元素将相关配置属性分组，以将结构添加到文件中
TargetFramework	此元素指定构建过程所针对的.NET Framework版本，必须在PropertyGroup元素中定义。默认值为netcoreapp2.0，以.NET Core 2.0为目标
ItemGroup	此元素将相关配置项分组，以将结构添加到文件中
Folder	此元素告诉MSBuild如何处理项目中的文件夹。列表中的元素告诉MSBuild在发布应用程序时应包含wwwroot文件夹
PackageReference	此元素用于指定NuGet包的依赖项，NuGet包可通过Include和Version属性进行标识。Microsoft.AspNetCore.All包用于提供对所有ASP.NET Core和MVC Framework功能的访问

14.2.1　将包添加到项目中

.csproj文件最重要的作用是列出项目所依赖的包。当Visual Studio检测到对.csproj文件进行的更改时，它会检查包的列表，下载新添加的内容，并删除所有不再需要的包。

随着ASP.NET Core 2的发布，ASP.NET Core MVC、MVC框架和Entity Framework Core所需的所有基本功能都包含在Microsoft.AspNetCore.All元数据包中。这是一个很方便的功能，避免了开始新项目时需要向项目添加一长串NuGet包的问题。

即便如此，也需要为第三方或高级功能添加NuGet包。有3种方法可以将包添加到项目中。第一种是选择Tools→NuGet Package Manager→Manage NuGet Packages for Solution，可以通过简单方便的界面管理NuGet包。如果你不熟悉.NET开发，那么这是最好的方法，因为它可以减少选择包时出错的可能性。

例如，要添加System.Net.Http包，这个包提供了创建（不仅仅是接收）HTTP请求的支持，可以转到包管理器的Browse部分，按名称搜索，并查看可用版本的完整列表，包括任何预发布版本，如图

14-2所示。

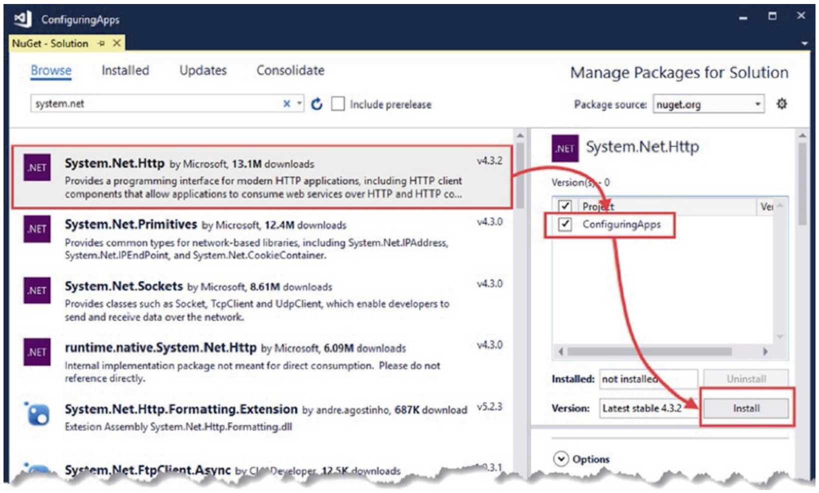


图14-2　使用NuGet包管理器添加包

第二种方法是选择所需的软件包和版本，然后选择需要软件包的项目，单击Install按钮，Visual Studio将下载软件包并更新.csproj文件。

最后一种方法是使用命令行添加包，但需要知道包的名称（理想情况下还需要知道版本）。代码清单14-6显示了在ConfiguringApps文件夹中运行的命令，用于将System.Net.Http包添加到项目中。

代码清单14-6　将包添加到项目中

```
dotnet add package System.Net.Http --version 4.3.2
```

NuGet包管理器和dotnet add package命令都会将PackageReference元素添加到.csproj文件中。如果愿意，可以编辑配置文件，通过手动添加PackageReference元素来添加包。这是最直接的方法，但需要注意避免错误输入包的名称或指定不存在的版本号。在代码清单14-7中，可以看到为了添加System.Net.Http包对.csproj文件所做的修改。

代码清单14-7　在ConfiguringApps文件夹下的InstallingApps.csproj文件中添加包

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
```

```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version
="4.3.2" />
  </ItemGroup>
```

PackageReference元素包含用于指定包名称的Include属性和用于指定版本号的Version属性。

14.2.2　将工具包添加到项目中

虽然能够以不同的方式添加常规包，但是某些包扩展了dotnet命令行工具的功能，这些包需要.csproj文件中的不同类型的元素，比如DotNetCliToolReference元素，而不是由应用程序直接使用的包所需的PackageReference元素。因此，只能通过直接编辑.csproj文件将这些包添加到项目中。

代码清单14-8显示了添加的包，允许使用本书

第一部分使用的dotnet ef命令创建和应用数据库迁移。

代码清单14-8　将工具包添加到ConfiguringApps文件夹下的InstallingApps.csproj文件中

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version
="4.3.2" />
    <DotNetCliToolReference Include="Microsoft.EntityFr
ameworkCore.Tools.DotNet"
        Version="2.0.0 " />
  </ItemGroup>

</Project>
```

在将工具包添加到项目中时，可以将

DotNetCliToolReference元素包含在与常规PackageReference元素相同的ItemGroup中，如代码清单14-8所示，或创建单独的ItemGroup元素。将更改保存到.csproj文件时，Visual Studio将下载并安装软件包，然后使用它们来配置dotnet命令行工具。

14.3　理解Program类

Program类定义在名为Program.cs的文件中，并提供运行应用程序的入口点。Program类为.NET提供了Main方法，可以执行Main方法来配置托管环境并选择完成ASP.NET Core应用程序的配置类。Program类的默认内容足以支持大多数项目的启动和运行，代码清单14-9显示了Visual Studio添加到项目中的默认代码。

代码清单14-9　ConfiguringApps文件夹下的Program.cs文件的默认内容

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

Main方法提供了所有.NET应用程序必须提供的入口点，以便运行时可以执行它们。Program类中的Main方法会调用BuildWebHost方法，BuildWebHost方法则负责配置ASP.NET Core。

BuildWebHost方法使用WebHost类定义的静态方法来配置ASP.NET Core。随着ASP.NET Core 2的发布，可通过使用CreateDefaultBuilder方法简化配置，该方法使用适合大多数项目的设置来配置ASP.NET Core。可调用UseStartup方法以标识特定于应用程序的配置类，但约定是使用名为Startup的类，详情将在本章后面介绍。Build方法处理所有配置并创建一个实现了IWebHost接口的对象，然后返回给Main方法，Main方法调用Run以开始处理HTTP请求。

深入配置细节

CreateDefaultBuilder方法是一种快速启动ASP.NET Core配置的便捷方法，但它确实隐藏了许多重要细节，如果需要更改应用程序的配置方式，这可能会是一个问题。代码清单14-10将CreateDefaultBuilder方法替换成了调用创建默认配

置的单条语句。

代码清单14-10　ConfiguringApps文件夹下的Program.cs文件中的详细配置语句

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {
            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    var env = hostingContext.HostingEnv
```

```
ironment;
                    config.AddJsonFile("appsettings.jso
n", optional: true,
                            reloadOnChange: true)
                        .AddJsonFile($"appsettings.{env
.EnvironmentName}.json",
                            optional: true, reloadOnCha
nge: true);

                    if (env.IsDevelopment()) {
                        var appAssembly =
                            Assembly.Load(new AssemblyN
ame(env.ApplicationName));
                        if (appAssembly != null) {
                            config.AddUserSecrets(appAs
sembly, optional: true);
                        }
                    }

                    config.AddEnvironmentVariables();

                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logg
ing) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.Ge
tSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                    .UseIISIntegration()
                    .UseDefaultServiceProvider((context
, options) => {
```

```
                        options.ValidateScopes =
                            context.HostingEnvironment.
IsDevelopment();
                    })
                    .UseStartup<Startup>()
                    .Build();
        }
    }
}
```

表14-4列出了添加到BuildWebHost方法的所有配置方法，并提供相应功能的简要说明。

表**14-4**　添加到**BuildWebHost**方法的所有配置方法

方法	描述
UseKestrel	用于配置Kestrel Web服务器
UseContentRoot	用于配置应用程序的根目录，根目录用于加载配置文件并提供静态内容，如图像、JavaScript和CSS
ConfigureAppConfiguration	用于为应用程序准备配置数据
AddUserSecrets	用于在代码文件之外存储敏感数据。这是一个有点棘手的方法，本书没有使用
ConfigureLogging	用于配置应用程序的日志记录

UseIISIntegration	用于启用与IIS和IIS Express的集成
UseDefaultServiceProvider	用于配置依赖注入
UseStartup	指定将用于配置ASP.NET的类

代码清单14-10已经解释了一些更复杂的语句。现在，删除一些配置语句，只保留基本配置，如代码清单14-11所示。

代码清单14-11　简化ConfiguringApps文件夹下的Program.cs文件中的配置

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {
```

```
        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

这些语句提供了适用于大多数ASP.NET Core MVC应用程序的基本配置。当解释其他语句的功能时，再把它们添加回来。

直接使用**Kestrel**

Kestrel是跨平台的Web服务器，旨在运行

ASP.NET Core应用程序。当使用IIS Express（Visual Studio提供的在开发期间使用的服务器）或完整版IIS（.NET应用程序的传统Web平台）运行ASP.NET Core应用程序时，会自动使用Kestrel。

如有需要，也可以直接运行Kestrel，这意味着可以在任何支持的平台上运行ASP.NET Core MVC应用程序，绕过IIS仅支持Windows平台的限制。有两种方法可以使用Kestrel运行应用程序。可以单击Visual Studio工具栏上IIS Express按钮右边缘的箭头，然后选择与项目名称匹配的选项。这将打开一个新的命令提示符并使用Kestrel运行应用程序。

也可以通过以下方法实现相同的效果：打开命令提示符，导航到包含应用程序配置文件的文件夹（包含.csproj文件的那个文件夹）并运行以下命令。

```
dotnet run
```

默认情况下，Kestrel服务器开始在端口5000上侦听HTTP请求。如果项目中有Properties/launchSettings.json文件，就从这个文件中读取应用程序的HTTP端口和环境。

14.4　了解Startup类

Program类负责启动应用程序，但最重要的配置工作是通过UseStartup方法委派的，如下所示：

```
...
.UseStartup<Startup>()
...
```

UseStartup方法依赖于type参数来标识用于配置ASP.NET Core的类。这个类的常规名称是Startup，Startup也是ASP.NET Core MVC项目模板（包括用

于为本章创建示例项目的Empty模板）使用的名称。

研究Startup类的工作原理可以帮助你深入了解HTTP请求的处理方式，以及MVC如何集成到ASP.NET Core平台的其余部分。

在本节中，将从最简单的Startup类开始，并添加功能以演示不同配置选项的效果，最后使用适合大多数MVC项目的配置。作为起点，代码清单14-12显示了Visual Studio添加到Empty项目中的Startup类，这里为该类设置了足够的功能以使ASP.NET Core能够处理HTTP请求。

代码清单14-12　Startup.cs文件的初始内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.Run(async (context) => {
                await context.Response.WriteAsync("Hell
o World!");
            });
        }
    }
}
```

Startup类定义了两个方法——ConfigureServices和Configure，它们用于设置应用程序所需的共享功能，并告诉ASP.NET Core应该如何使用它们。

当应用程序启动时，ASP.NET Core会创建一个新的Startup实例并调用ConfigureServices方法，以便应用程序可以创建服务。正如14.4.1节所解释的，服务是为应用程序其他部分提供功能的对象。

以上描述过于模糊，因为服务可以用来提供任何功能。创建服务后，ASP.NET将调用Configure方法。Configure方法的目的是设置请求管道，请求管道是一组组件——称为中间件——用于处理传入的HTTP请求并为它们生成响应。其中解释请求管道的工作原理，并演示如何在14.4.2节中创建中间件。图14-3显示了ASP.NET使用Startup类的方式。

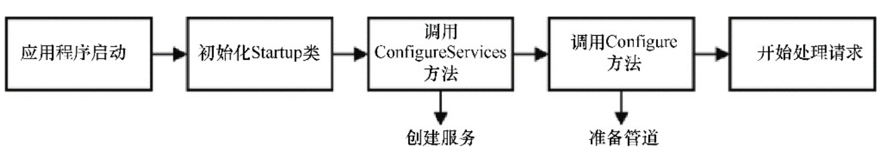


图14-3　ASP.NET如何使用Startup类配置应用程序

让Startup类为所有请求返回相同的“Hello，

World”消息是没有意义的，所以在详细解释类中的方法之前，需要先启用MVC，如代码清单14-13所示。

代码清单14-13　在ConfiguringApps文件夹下的Startup.cs文件中启用MVC

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

后面的章节会解释这些修改，现在已经有足够的基础设施代码来处理HTTP请求并使用控制器和视图生成响应了。如果运行应用程序，你将看到图14-4所示的输出。

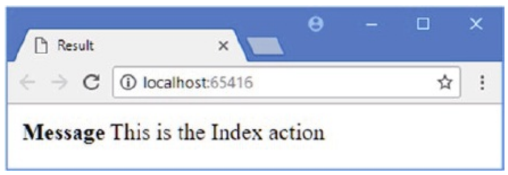


图14-4　启用MVC的效果

请注意，输出的内容尚未设置样式。代码清单14-13中的最小配置不支持提供静态内容，如CSS样式表和JavaScript文件，因此当Index.cshtml视图文件中的link元素请求Bootstrap CSS样式时，应用程序无法处理，导致浏览器无法获取所需的CSS样式。14.4.4节将介绍如何解决这个问题。

14.4.1　了解ASP.NET服务

ASP.NET Core调用Startup.ConfigureServices方法，以便应用程序可以设置所需的服务。服务是指为应用程序的其他部分提供功能的对象。如上所述，这样的描述过于模糊，因为服务可以执行应用程序所需的任何操作。举例来说，在项目中添加Infrastructure文件夹，并在其中添加一个名为UptimeService.cs的类文件，用来定义代码清单14-14所示的类。

代码清单14-14　Infrastructure文件夹下的UptimeService.cs文件的内容

```
using System.Diagnostics;

namespace ConfiguringApps.Infrastructure {

    public class UptimeService {
        private Stopwatch timer;

        public UptimeService() {
            timer = Stopwatch.StartNew();
        }

        public long Uptime => timer.ElapsedMilliseconds
;
```

```
    }
}
```

创建UptimeService类时，UptimeService构造函数会启动一个计时器，以跟踪应用程序的运行时间。这是一个很好的服务示例，因为它提供了可以在应用程序的其他部分使用的功能，并且可以在应用程序启动时创建。

ASP.NET服务使用Startup类的ConfigureServices方法来注册，在代码清单14-15中，可以看到如何注册UptimeService类。

代码清单14-15　在ConfiguringApps文件夹下的Startup.cs文件中注册自定义服务

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

作为参数，ConfigureServices方法会接收实现了IServiceCollection接口的对象。可使用在IServiceCollection接口上调用的扩展方法注册服务，这些扩展方法指定了不同的配置选项。第18章将描述可用于创建服务的选项，但目前使用了AddSingleton方法，这意味着整个应用程序将共享一个UptimeService对象。

服务与依赖注入功能密切相关，后者允许控制器等组件轻松获取服务，这将在第18章深入介绍。在Startup.ConfigureServices中注册的服务可以通过创建接收参数的构造函数来访问，参数需要为请求的服务类型。代码清单14-16显示了添加到Home控制器的构造函数，以接收代码清单14-15中创建的共享的UptimeService对象。这里还更新了控制器的Index操作方法，以便在生成的视图数据中包含服务的Update属性值。

代码清单14-16　在Controllers文件夹下的HomeController.cs文件中访问服务

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;

        public HomeController(UptimeService up) => upti
me = up;
```

```
        public ViewResult Index()
            => View(new Dictionary<string, string> {
                ["Message"] = "This is the Index action
",
                ["Uptime"] = $"{uptime.Uptime}ms"
            });
    }
}
```

当MVC需要Home控制器类的实例来处理HTTP请求时，它会检查HomeController构造函数并发现需要UptimeService对象。然后，MVC检查已在Startup类中配置的服务集，发现已经配置过UptimeService，这样就可以将单例UptimeService对象用于所有请求，并在创建HomeController时传递该对象作为构造函数参数。

虽然可以使用更复杂的方式注册和使用服务，但此例演示了服务背后的核心思想，并说明了如何在Startup类中定义服务，以便定义整个应用程序中使用的功能或数据。

如果运行应用程序并请求默认URL，你将看到一个响应，包括自应用程序启动以来的毫秒数，毫秒数是从Startup类中创建的UptimeService对象获得的，如图14-5所示（严格来说，这是自创建UptimeService服务对象以来的时间，但这与应用程序的启动时间非常接近，不会对本章的目的产生任何影响）。

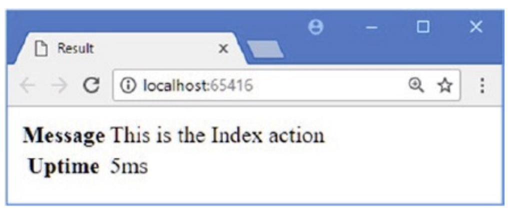


图14-5　使用简单服务获得的响应

每次收到对默认URL的请求时，MVC都会创建一个新的HomeController对象，并为其提供共享的UptimeService对象作为构造函数参数。这允许Home控制器访问应用程序的正常运行时间，而不用关心如何提供或实现此信息。

了解内置的MVC服务

像MVC这样的复杂软件包使用了许多服务。一些供内部使用，另一些为开发人员提供功能。包定义扩展方法，在单个方法调用中设置它们所需的所有服务。对于MVC，这个方法名为AddMvc，它是添加到Startup类以使MVC工作的两个方法之一。

```
...
public void ConfigureServices(IServiceCollection servic
es) {
    services.AddSingleton<UptimeService>();
    services.AddMvc();
}
...
```

这个方法设置MVC所需的每个服务，无须在ConfigureServices方法中使用大量单个服务。

注　意

Visual Studio的智能感知功能将显示可以在ConfigureServices方法中的IServiceCollection对象上调用的其他扩展方法的长列表。其中一些方法（如AddSingleton和AddScoped）用于以不同方式注册服务，其他方法（如AddRouting或AddCors）用于添加已由AddMvc方法使用的单个服务。结果是，对于大多数应用程序，ConfigureServices方法仅包含少量自定义服务（对AddMvc方法的调用）以及一些可选的语句来配置内置服务，14.6节将对此进行描述。

14.4.2　了解ASP.NET中间件

在ASP.NET Core中，中间件是指用来组合请求管道的组件。请求管道像链一样排列，当新请求到

达时，新请求被传递给链中的第一个中间件。这个中间件检查请求并决定是否处理并生成响应，也可将请求传递给链中的下一个中间件。处理完请求后，将返回到客户端的响应沿链传回，这允许所有前面的中间件检查或修改响应。

中间件的工作方式似乎有点奇怪，但它允许应用程序组合在一起的方式具有很大的灵活性。了解如何使用中间件来构建应用程序非常重要，尤其是在没有得到预期的响应时。为了解释中间件系统的工作原理，下面创建一些自定义组件来演示你将遇到的4种类型的中间件。

1．创建内容生成中间件

最重要的中间件类型用于为客户端生成内容，MVC就属于这种类型。为了创建比MVC简单的内容生成中间件，在Infrastructure文件夹中添加一个

名为ContentMiddleware.cs的类，用来定义代码清单14-17所示的类。

代码清单14-17　Infrastructure文件夹下的ContentMiddleware.cs文件的内容

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ContentMiddleware {
        private RequestDelegate nextDelegate;

        public ContentMiddleware(RequestDelegate next)
=> nextDelegate = next;
        public async Task Invoke(HttpContext httpContex
t) {
            if (httpContext.Request.Path.ToString().ToL
ower() == "/middleware") {
                await httpContext.Response.WriteAsync(
                "This is from the content middleware",
Encoding.UTF8);
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

中间件既不实现接口，也不从公共基类派生。相反，它们定义了一个构造函数，这个构造函数接收RequestDelegate对象并定义了Invoke方法。RequestDelegate对象表示链中的下一个中间件，当ASP.NET收到HTTP请求时调用Invoke方法。

HTTP请求和返回给客户端的响应信息可通过Invoke方法的HttpContext参数提供。第17章将描述HttpContext类及其属性，但是在本章中，只需要知道代码清单14-17中的Invoke方法能够检查HTTP请求并验证请求是否已发送到/middleware URL就可以了。如果有，就向客户端发送简单的文本响应；如果使用了不同的URL，就将请求转发到链中的下一个中间件。

请求管道是在Startup类的Configure方法中设置的。在代码清单14-18中，从示例应用程序中删除了MVC方法，并使用ContentMiddleware类作为管道

中的唯一组件。

代码清单14-18　在ConfiguringApps文件夹下的Startup.cs文件中使用自定义中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

可使用Configure方法中的UseMiddleware扩展方法注册自定义中间件。UseMiddleware方法使用类型参数来指定中间件类。这样ASP.NET Core就可以构建一个将要使用的所有中间件的列表，然后将它们实例化以创建链。如果运行应用程序并请求/middleware URL，你将看到图14-6所示的结果。



图14-6　从自定义中间件生成内容

图14-7展示了使用ContentMiddleware类创建的中间件管道。当ASP.NET Core收到HTTP请求时，它会将请求传递给Startup类中注册的唯一中间件。如果URL是/middleware，组件将生成结果，返回给ASP.NET Core并发送给客户端。

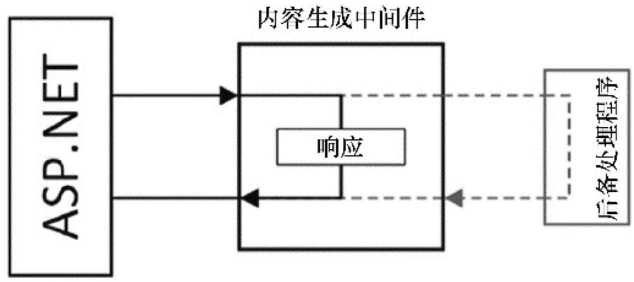


图14-7　示例中间件管道

如果URL不是/middleware，ContentMiddleware类就将请求传递给链中的下一个中间件。但由于没有其他组件，因此请求在创建管道时到达ASP.NET Core提供的后备处理程序，后备处理程序在另一个方向上沿着管道发回请求（如果能看到其他类型的中间件如何工作，这个过程就容易理解了）。

并非只有控制器能够使用ConfigureServices方法中设置的服务。ASP.NET Core检查中间件类的构造函数，并使用服务为已定义的任何参数提供值。代码清单14-19向ContentMiddleware类的构造函数添加了一个参数，以告诉ASP.NET Core需要一个UptimeService对象。

代码清单14-19　在Infrastructure文件夹下的ContentMiddleware.cs文件中使用服务

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ContentMiddleware {
        private RequestDelegate nextDelegate;
        private UptimeService uptime;

        public ContentMiddleware(RequestDelegate next,
UptimeService up) {
            nextDelegate = next;
            uptime = up;
        }

        public async Task Invoke(HttpContext httpContex
t) {
            if (httpContext.Request.Path.ToString().ToL
ower() == "/middleware") {
                await httpContext.Response.WriteAsync(
                    "This is from the content middlewar
e "+
                        $"(uptime: {uptime.Uptime}ms)",
 Encoding.UTF8);
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

能够使用服务意味着中间件可以共享通用功能并避免代码重复。运行应用程序并请求/middleware URL，你将看到图14-8所示的输出。



图14-8　在自定义中间件中使用服务

2．创建短路中间件

下一类中间件在请求到达内容生成中间件之前拦截它们，用来对管道处理过程进行短路，这通常是为了优化性能。代码清单14-20显示了添加到Infrastructure文件夹中的名为ShortCircuitMiddleware.cs的类文件的内容。

代码清单14-20　Infrastructure文件夹下的ShortCircuitMiddleware.cs文件的内容

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ShortCircuitMiddleware {
        private RequestDelegate nextDelegate;

        public ShortCircuitMiddleware(RequestDelegate n
ext) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContex
t) {
            if (httpContext.Request.Headers["User-Agent
"]
                    .Any(h => h.ToLower().Contains("edg
e"))) {
                httpContext.Response.StatusCode = 403;
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

这个中间件检查请求的User-Agent标头，浏览器使用该标头标识自身。使用User-Agent标头来标识特定的浏览器并不可靠，无法在实际应用程序中使用，但对此例来说已经足够了。

使用术语“短路”是因为这种类型的中间件并不总是将请求转发到链中的下一个中间件。在这种情况下，如果User-Agent标头包含术语edge，那么组件会将状态码设置为403-Forbidden，并且不将请求转发到下一个中间件。由于请求被拒绝，因此再由其他中间件处理请求是没有意义的，会不必要地消耗系统资源。相反，将请求处理提前终止，向客户端发送403响应。

中间件按照Startup类中设置的顺序接收请求，这意味着必须在内容生成中间件之前设置短路中间件，如代码清单14-21所示。

代码清单14-21　在ConfiguringApps文件夹下的Startup.cs文件中注册短路中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
```

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMiddleware<ShortCircuitMiddleware>();

            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

如果使用Microsoft Edge浏览器运行应用程序并请求任何URL，那么你将看到403错误。ShortCircuitMiddleware组件会忽略来自其他浏览器的请求，并将请求传递给链中的下一个中间件，这意味着当请求的URL为/middleware时将生成响应。将短路中间件添加到中间件管道中，如图14-9所

示。

3．创建请求编辑中间件

下一类中间件不会生成响应。相反，它们会在请求到达链中的其他中间件之前更改请求。这种中间件主要用于平台集成，还可用于准备请求，以便后续组件更容易处理它们。作为演示，将BrowserTypeMiddleware.cs文件添加到Infrastructure文件夹中，并用它定义代码清单14-22所示的中间件。

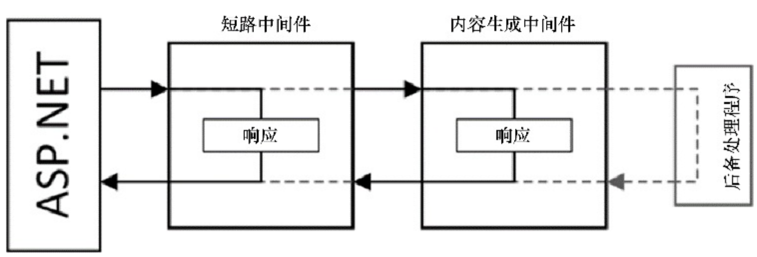


图14-9　在中间件管道中添加短路中间件

代码清单14-22　Infrastructure文件夹下的BrowserTypeMiddleware.cs文件的内容

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class BrowserTypeMiddleware {
        private RequestDelegate nextDelegate;
        public BrowserTypeMiddleware(RequestDelegate ne
xt) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContex
t) {
            httpContext.Items["EdgeBrowser"]
                = httpContext.Request.Headers["User-Age
nt"]
                    .Any(v => v.ToLower().Contains("edg
e"));
            await nextDelegate.Invoke(httpContext);
        }
    }
}
```

这个中间件检查请求的User-Agent标头并查找术语edge，这表明请求可能是使用Microsoft Edge浏览器进行的。HttpContext对象通过Items属性提供了一个字典，该字典用于在组件之间传递数据，请求将标头的搜索结果与键EdgeBrowser一起存储。

为了演示中间件如何协作，代码清单14-23显示了ShortCircuitMiddleware类，它拒绝来自Microsoft Edge浏览器的请求，并根据BrowserTypeMiddleware组件生成的数据做出决策。

代码清单14-23　与ShortCircuitMiddleware.cs文件中的另一个中间件协作

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ShortCircuitMiddleware {
        private RequestDelegate nextDelegate;

        public ShortCircuitMiddleware(RequestDelegate n
ext) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContex
t) {
            if (httpContext.Items["EdgeBrowser"] as boo
l? == true) {
                httpContext.Response.StatusCode = 403;
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
```

```
    }
}
```

就本质而言，编辑请求中间件需要放置在与之合作的或者依赖于它们的中间件之前。在代码清单14-24中，将BrowserTypeMiddleware类注册为中间件管道中的第一个中间件。

代码清单14-24　在ConfiguringApps文件夹下的Startup.cs文件中注册中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>()
;
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

将中间件放置在中间件管道的开始处可确保在其他中间件收到请求之前已对请求进行了修改，如图 14-10所示。

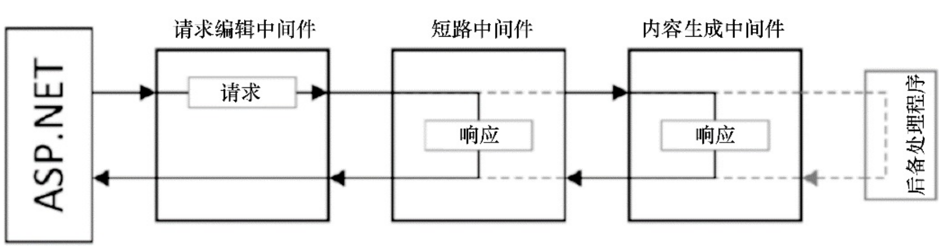


图14-10　将请求编辑组件添加到中间件管道中

4．创建响应编辑中间件

最后一类中间件用于对中间件管道中其他中间件生成的响应进行操作，这对于记录请求及响应的

详细信息或处理错误非常有用。代码清单14-25显示了ErrorMiddleware.cs文件的内容，将这个文件添加到Infrastructure文件夹中以演示此类中间件。

代码清单14-25　Infrastructure文件夹下的ErrorMiddleware.cs文件的内容

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ErrorMiddleware {
        private RequestDelegate nextDelegate;

        public ErrorMiddleware(RequestDelegate next) {
            nextDelegate = next;
        }

        public async Task Invoke(HttpContext httpContex
t) {
            await nextDelegate.Invoke(httpContext);

            if (httpContext.Response.StatusCode == 403)
 {
                await httpContext.Response
                    .WriteAsync("Edge not supported", E
ncoding.UTF8);
            } else if (httpContext.Response.StatusCode
```

```
== 404) {
                await httpContext.Response
                    .WriteAsync("No content middleware
response", Encoding.UTF8);
            }
        }
    }
}
```

这个中间件在通过中间件管道并且生成响应之前对请求不感兴趣。如果状态码为 403 或 404，就向响应添加描述性消息。所有其他响应都被忽略。代码清单14-26显示了如何在Startup类中注册响应编辑中间件。

代码清单14-26　在Startup.cs文件中注册响应编辑中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
```

```
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMiddleware<ErrorMiddleware>();
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>()
;
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

提　示

你可能想知道产生404状态码的原因，因为它不是由这里创建的3个中间件中的任何一个设置的。答案是，这是当请求进入管道时由ASP.NET配

置的响应，如果没有中间件更改响应，返回给客户端的状态码就是404。

我们注册了ErrorMiddleware类，它占据中间件管道中的第一个位置。对于仅需要处理响应的中间件，这可能看起来很奇怪，但是在链的开始位置注册中间件可确保能够处理任何其他中间件生成的响应，如图14-11所示。如果放置在管道的后面位置，那么只能处理由某些其他中间件生成的响应。

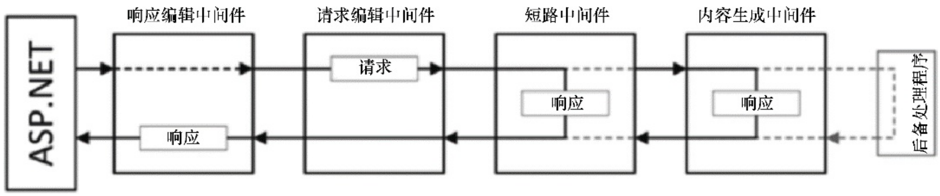


图14-11　将响应编辑中间件添加到中间件管道中

可以通过启动应用程序并请求除/middleware URL外的任何URL来查看新添加的中间件的效果。

结果将显示图14-12所示的错误消息。



图14-12 编辑其他中间件时的响应

14.4.3 了解如何调用Configure方法

ASP.NET Core平台会在调用之前检查Configure方法，并获取其参数列表，该列表由ConfigureServices方法中设置的服务或表14-5中显示的特殊服务（可用作Configure方法的参数）提供。

表14-5 特殊服务

特殊服务	描述
IApplicationBuilder	定义了设置应用程序中间件管道所需的功能
IHostingEnvironment	定义了区分不同类型环境（如开发环境和生产环境）所需的功能

1．使用Application Builder

虽然不必为Configure方法定义任何参数，但大多数Startup类至少使用IApplicationBuilder，因为它允许创建中间件管道。对于自定义中间件来说，UseMiddleware扩展方法用于注册类。复杂的内容生成中间件包提供了单一方法以在一个步骤中设置所有的中间件，就像它们提供单一方法来定义使用的服务一样。对于MVC来说，可以使用两种扩展方法，如表14-6所示。

表**14-6**　两种扩展方法

扩展方法	描述
UseMvcWithDefaultRoute	使用默认路由设置MVC中间件
UseMvc	使用Lambda表达式指定的自定义路由配置设置MVC中间件

路由是将请求URL映射到控制器并由应用程序

定义操作的过程，第15章和第16章将详细描述路由。UseMvcWithDefaultRoute扩展方法对于开始使用MVC很有用，但是大多数应用程序会调用UseMvc扩展方法，即使结果是显式定义了由UseMvcWithDefaultRoute扩展方法创建的相同的路由配置，如代码清单 14-27所示。这使得其他开发人员能够很容易理解应用程序使用的路由配置，并且以后可以轻松添加新的路由（几乎所有应用程序在某些时候都需要）。

代码清单14-27　在ConfiguringApps文件夹下的Startup.cs文件中设置MVC中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
```

```
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseMiddleware<ErrorMiddleware>();
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>()
;
            app.UseMiddleware<ContentMiddleware>();

            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

由于MVC设置了内容生成中间件，因此UseMvc扩展方法必须在注册了所有其他中间件之后调用。为了准备MVC所依赖的服务，必须在ConfigureServices方法中调用AddMvc扩展方法。

2．使用托管环境

IHostingEnvironment接口使用表14-7所示的属性提供有关运行应用程序的托管环境的一些基本但非常重要的信息。

表**14-7　IHostingEnvironment**属性

属性	描述
ApplicationName	返回应用程序的名称，该名称由托管平台设置
EnvironmentName	返回描述当前环境的字符串
ContentRootPath	返回包含应用程序的内容文件和配置文件的路径
WebRootPath	返回一个字符串，该字符串指定包含应用程序静态内容的文件夹，通常是wwwroot文件夹
ContentRootFileProvider	返回一个实现了IFileProvider接口的对象，该对象可用于从ContentRootPath属性指定的文件夹中读取文件
WebRootFileProvider	返回一个实现了IFileProvider接口的对象，该对象可用于从WebRootPath属性指定的文件夹中读取文件

ContentRootPath和WebRootPath属性很有意思，但大多数应用程序中并不需要这两个属性，因为有一个内置的中间件可用于传递静态内容。

比较重要的属性是EnvironmentName，它允许根据应用程序运行的环境修改应用程序的配置。有3种常规环境——开发环境、暂存环境和生产环境。

使用名为ASPNETCORE_ENVIRONMENT的环境变量设置当前托管环境。要设置环境变量，请从Visual Studio的Project菜单中选择ConfiguringApps Properties，然后切换到Debug选项卡。双击环境变量的Value字段，默认情况下设置为Development，将之更改为Staging，如图14-13所示。保存更改以使新环境生效。

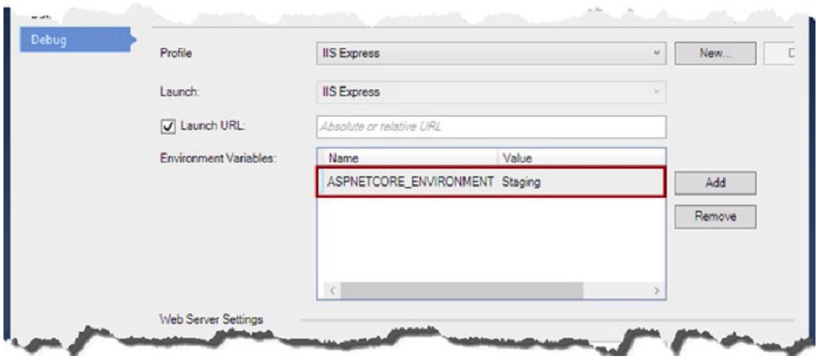


图14-13　设置托管环境的名称

提　示

环境名称不区分大小写，因此Staging和staging可视为同一环境。虽然Development、Staging和Production是传统的环境名称，但你可以使用自己喜欢的任何名称。例如，如果项目中有多个开发人员，并且每个开发人员都需要不同的配置设置，那么这可能很有用。有关如何处理环境配置之间复杂

差异的详细信息，请参阅14.7节。

在Configure方法中，可以通过读取IHostingEnvironment.EnvironmentName属性或使用对IHostingEnvironment对象进行操作的扩展方法之一来确定正在使用的托管环境，如表14-8所示。

表**14-8　IhostingEnvironment**的扩展方法

扩展方法	描述
IsDevelopment()	如果托管环境的名称为Development，返回true
IsStaging()	如果托管环境的名称是Staging，返回true
IsProduction()	如果托管环境的名称为Production，返回true
IsEnvironment（env）	如果托管环境的名称与env参数匹配，返回true

扩展方法用于更改中间件管道中的中间件集

合，以定制应用程序到不同托管环境的行为。代码清单14-28使用一种扩展方法来确保本章前面创建的自定义中间件仅出现在Development托管环境的管道中。

代码清单14-28　在ConfiguringApps文件夹下的Startup.cs文件中使用托管环境

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
```

```
            if (env.IsDevelopment()) {
                app.UseMiddleware<ErrorMiddleware>();
                app.UseMiddleware<BrowserTypeMiddleware
>();
                app.UseMiddleware<ShortCircuitMiddlewar
e>();
                app.UseMiddleware<ContentMiddleware>();
            }

            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

这3个自定义中间件不会添加到使用当前配置的管道中，当前配置已将托管环境设置为Staging。如果运行应用程序并请求/middleware URL，你将收到404-Not Found错误，因为唯一可用的中间件是使用UseMvc扩展方法设置的，它们没有可用于处理该URL的控制器。

注　意

一旦测试了更改托管环境的效果，就请务必更改回Development；否则，本章其余部分的示例将无法正常工作。

14.4.4　添加其他中间件

有一组常用的中间件在大多数MVC项目中很有用，本书的示例中也使用了这些中间件。下面将这些中间件添加到请求管道中并解释它们的工作原理。

1．启用异常处理

即使是精心编写的应用程序也会遇到异常，因此适当地处理它们非常重要。代码清单14-29添加了一些中间件，用于处理请求管道发生的异常。这里还删除了自定义中间件，以便可以专注于MVC。

代码清单14-29　在ConfiguringApps文件夹下的Startup.cs文件中添加异常处理中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
```

```
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

UseStatusCodePages方法将描述性消息添加到不包含内容的响应中，例如404-Not Found响应，这可能很有用，因为并非所有浏览器都向用户显示自己的消息。

UseDeveloperExceptionPage方法设置一个错误处理中间件，这个中间件在响应中显示异常的详细信息，包括异常跟踪。这些信息不应该向用户显

示，因此只能在使用IHostingEnvironment对象检测到的开发托管环境中调用UseDeveloperExceptionPage方法。

对于Staging或Production环境，使用UseExceptionHandler方法。该方法设置错误处理，允许显示自定义错误消息，不会显示应用程序的内部错误细节。UseExceptionHandler方法的参数是客户端应重定向到的URL，以便接收错误消息。这可以是应用程序提供的任何URL，但惯例是使用/Home/Error。

在代码清单14-30中，添加了根据需要为Home控制器的Index操作生成异常的功能，并添加了Error操作，以处理UseExceptionHandler生成的请求。

代码清单14-30　在Controllers文件夹下的HomeController.cs文

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;

        public HomeController(UptimeService up) => upti
me = up;

        public ViewResult Index(bool throwException = f
alse) {
            if (throwException) {
                throw new System.NullReferenceException
();
            }
            return View(new Dictionary<string, string> {

                ["Message"] = "This is the Index action
",
                ["Uptime"] = $"{uptime.Uptime}ms"
            });
        }

        public ViewResult Error() => View(nameof(Index),

            new Dictionary<string, string> {
                ["Message"] = "This is the Error action
"});
            }
```

```
}
```

对Index操作的更改依赖于第26章描述的模型绑定功能，以从请求中获取throwException的值。如果throwException为true，Index操作抛出NullReferenceException；如果为false，就正常执行。

Error操作使用Index视图显示简单消息。可以通过运行应用程序并请求/Home/Index?throwException=true URL来查看不同异常处理中间件的效果。查询字符串提供了 Index 操作参数的值，你看到的响应将取决于托管环境的名称。图14-14显示了UseDeveloperExceptionPage（用于开发托管环境）和UseExceptionHandler中间件（用于所有其他托管环境）生成的输出结果。

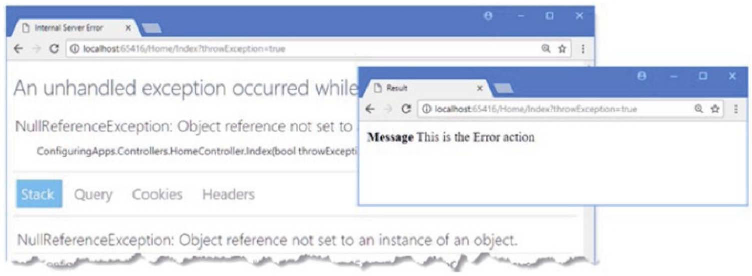


图14-14　输出结果

开发人员异常页面提供了异常的详细信息，以及查看堆栈跟踪和导致异常的请求的选项。相比之下，用户异常页面应该仅用于表示发生了错误。

2．启用浏览器链接

第6章介绍了浏览器链接功能，并演示了如何在开发过程中用来管理浏览器。浏览器链接的服务器端部分是作为中间件实现的，必须作为应用程序配置的一部分添加到Startup类中；否则，Visual Studio集成将无法工作。浏览器链接仅在开发期间有用，不应在暂存环境或生产环境中使用，因为还会编辑其他中间件生成的响应，并插入JavaScript代

码。打开到服务器端的HTTP连接，以接收重新加载的通知。在代码清单14-31中，可以看到如何仅为开发托管环境调用已注册中间件的UseBrowserLink方法。

代码清单14-31　在ConfiguringApps文件夹下的Startup.cs文件中启用浏览器链接

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
```

```
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

3．启用静态内容

最后一个中间件对大多数项目很有用，它提供了对wwwroot文件夹中文件的访问，使应用程序可以包含图像、JavaScript文件和CSS。使用UseStaticFiles方法添加一个组件，用于将静态文件的请求管道短路，如代码清单14-32所示。

代码清单14-32　在ConfiguringApps文件夹下的Startup.cs文件

中启用静态内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
```

```
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

无论托管环境如何，通常都需要静态内容，这就是为所有环境调用UseStaticFiles方法的原因。这意味着Index视图中的link元素将正常工作，并允许浏览器加载Bootstrap CSS样式表。可以通过启动应用程序来查看效果，如图 14-15所示。

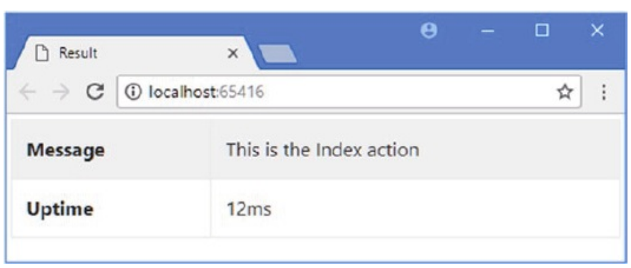


图14-15　启用静态内容

14.5　配置应用程序

某些配置数据经常更改，例如，当应用程序从

开发环境转移到生产环境时，数据库服务器需要不同的信息。ASP.NET Core并不在Startup类中对此信息进行硬编码，而是从一系列更容易更改的源端提供配置数据，例如环境变量、命令行参数以及使用JavaScript Object Notation（JSON）编写的文件。

配置数据通常是自动处理的，但由于已经替换了Program类中的默认设置，因此需要显式添加获取数据的代码，并使其可用于应用程序的其他部分，如代码清单14-33所示。

代码清单14-33　在ConfiguringApps文件夹下的Program.cs文件中加载配置数据

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;
```

```
namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    config.AddJsonFile("appsettings.jso
n",
                        optional: true, reloadOnChange:
 true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

ConfigureAppConfiguration方法用于处理配置

数据，其参数是WebHostBuilderContext对象和实现了IConfigurationBuilder接口的对象。WebBostBuilderContext类定义了表14-9中描述的属性。

表**14-9**　由**WebBostBuilderContext**类定义的属性

属性	描述
HostingEnvironment	返回一个实现了IHostingEnvironment接口的对象，并提供有关运行应用程序的托管环境的信息。有关详细信息
Configuration	返回一个实现了IConfiguration接口的对象，该接口提供对应用程序中配置数据的只读访问

IConfigurationBuilder接口用于为应用程序的其余部分准备配置数据，这通常使用扩展方法完成。表14-10描述了代码清单14-33中用于添加配置数据的3个扩展方法。

表**14-10**　用于添加配置数据的扩展方法

扩展方法	描述
AddJsonFile	用于从JSON文件（例如appsettings.json）加载配置数据
AddEnvironmentVariables	用于从环境变量加载配置数据
AddCommandLine	用于从启动应用程序的命令行参数加载配置数据

在代码清单14-33所示的用于加载配置数据的3个方法中，最有趣的是AddJsonFile方法。该方法的参数指定了文件名、文件是否可选以及在文件更改时是否应重新加载配置数据：

```
...
config.AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true);
...
```

以上代码指定了一个名为appsettings.json的文件，这是JSON配置文件的常规名称。appsettings.json文件是可选的，也就是说，如果文件不存在，不会引发异常，并且将监视更改和自动

刷新配置数据。

重新加载配置数据

ASP.NET Core配置系统支持在配置文件更改时重新加载数据。某些内置中间件（如日志记录系统）支持此功能，这意味着可以在运行时更改日志记录级别，而无须重新启动应用程序。你也可以在自定义中间件中包含类似功能。

但仅仅因为支持这个功能而使用它并不意味着就是切合实际的。在生产系统中更改配置文件是导致停机的原因之一，很容易产生输入错误并导致创建错误的配置。即使成功进行了更改，也可能出现无法预料的后果，例如日志数据填满磁盘或导致性能下降。

建议避免实时编辑，并确保在部署到生产环境之前将所有更改推送到标准测试流程中。针对实时运行的系统来诊断问题可能很容易，但结果可能并不会尽如人意。如果你正在编辑生产环境中的配置文件，那么应该慎重考虑是否要将一个小问题变成一个更大的问题。

14.5.1 创建JSON配置文件

appsettings.json文件的最常见用途是存储数据库连接字符串和日志记录设置，但也可以存储应用程序所需的任何数据。

要查看配置系统的工作方式，请将名为appsettings.json的新JSON文件添加到项目的根文件夹中，内容如代码清单14-34所示。

代码清单14-34　ConfiguringApps文件夹下的appsettings.json文件的内容

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
}
```

可以为JSON格式定义配置的结构。以上代码中的JSON内容定义了名为ShortCircuitMiddleware的配置类别，其中包含名为EnableBrowserShortCircuit的配置属性，它被设置为true。

JSON中的引号和逗号

如果不熟悉JSON，那么值得花一些时间阅读JSON网站上的规范。JSON格式易于使用，并且大多数平台对生成和解析JSON数据提供了良好支持，包括MVC应用程序和使用简单Javascript API的

客户端。事实上，大多数MVC开发人员根本不会直接与JSON打交道，只有在配置文件中才需要手动编码JSON。

关于JSON有两个很多开发人员容易遇到的陷阱，虽然仍然需要花时间阅读规范，但是当Visual Studio或ASP.NET Core无法解析JSON文件的时候，了解最常见的问题会让你更容易找到原因。下面是对appsettings.json文件的补充，显示了两个最常见的问题：

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  mysetting : [ fast, slow ]
}
```

首先，JSON的几乎所有内容都是用引号括起来的。如果正在编写C#代码并期望在没有引号的情况下接收属性的名称和值，就会很容易忘记这一

点。在JSON中，所有除布尔值和数字之外的其他内容必须用引号括起来，如下所示：

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  "mysetting" : [ "fast", "slow"]
}
```

其次，在向对象的JSON描述添加一个新的属性时，必须记住在上一个大括号字符的后面添加一个逗号，如下所示：

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
  "mysetting" : [ "fast", "slow"]
}
```

即使这个错误已高亮显示也很难看出差异——这就是为什么这类错误如此常见——但这里已经在关闭ShortCircuitMiddleware部分的“}”后添加了一个

逗号。另外要小心，引号后没有其他节点也是非法的。如果因为更改JSON导致问题发生，那么首先要检查这两个错误。

14.5.2　使用配置数据

Startup类可以通过使用IConfiguration参数定义的构造函数来访问配置数据。在Program类中调用UseStartup方法时，会使用ConfigureAppConfiguration准备的配置数据创建Startup对象。代码清单14-35向Startup类添加了构造函数，并显示了如何访问配置数据。

代码清单14-35　在ConfiguringApps文件夹下的Startup.cs文件中接收和使用配置数据

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Configuration;

namespace ConfiguringApps {
    public class Startup {

        public Startup(IConfiguration configuration) {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {

            if ((Configuration.GetSection("ShortCircuit
Middleware")?
                    .GetValue<bool>("EnableBrowserShort
Circuit")).Value) {
                    app.UseMiddleware<BrowserTypeMiddle
ware>();
                    app.UseMiddleware<ShortCircuitMiddl
eware>();
            }

            if (env.IsDevelopment()) {
```

```
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

构造函数接收IConfiguration对象并将其分配给名为Configuration的属性，然后可以使用该属性访问配置数据，这些配置数据已从环境变量、命令行和appsettings.json文件加载。

要获取值，可以将数据结构导航到所需的配置部分，这部分由实现了IConfiguration接口的另一个对象表示，该接口提供了可用于IConfigurationRoot的成员，如表14-11所示。

表**14-11**　可用于**IConfigurationRoot**的成员

成员	描述
[key]	该索引器用于获取特定键的字符串值
GetSection(name)	该方法返回一个IConfiguration对象，该对象表示配置数据的一个节点
GetChildren()	该方法返回IConfiguration对象的子节点条目，用来表示当前配置对象的子节点

还有一些扩展方法可用于对IConfiguration对象进行操作以获取值并将其从字符串转换为其他类型，如表14-12所示。

表**14-12**　**IConfiguration**接口的扩展方法

扩展方法	描述
GetValue<T>(keyName)	获取与指定键关联的值，并尝试将其转换为类型T
GetValue<T>(keyName,defaultValue)	获取与指定键关联的值，并尝试将其转换为类型T。如果配置数据中的键没有值，就使用默认值

重要的是，不要假设配置值一定会被指定。可使用null条件运算符来确保在尝试获取EnableBrowser- ShortCircuit值之前已获取到ShortCircuitMiddleware部分。结果是，只有在定义了ShortCircuitMiddleware/EnableBrowserShortCircuit值并将其设置为true时，才会将自定义中间件添加到请求管道中。

14.5.3 配置日志记录

ASP.NET Core提供对捕获和处理日志记录数据的支持，并且已内置许多中间件以生成日志记录消息。大多数项目会自动设置日志记录，但由于在Program类中使用了单独的配置语句，因此需要添加代码清单14-36所示的语句来设置日志记录功能。

代码清单14-36 在ConfiguringApps文件夹下配置Program.cs文

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    config.AddJsonFile("appsettings.jso
n",
                        optional: true, reloadOnChange:
 true);
                    config.AddEnvironmentVariables();
```

```
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logg
ing) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.Ge
tSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

ConfigureLogging方法使用Lambda函数设置日志记录系统，Lambda函数接收WebHostBuilderContext对象（可参考本章前面的内容）和实现了ILoggingBuilder接口的对象。ILoggingBuilder接口运行一组扩展方法来配置日志记录系统，如表14-13所示。

表**14-13　ILoggingBuilder**接口的扩展方法

扩展方法	描述
AddConfiguration	用于使用从appsettings.json文件、命令行或环境变量加载的配置数据来配置日志记录系统
AddConsole	将日志消息发送到控制台，这在使用dotnet run命令启动应用程序时非常有用
AddDebug	当Visual Studio调试器运行时，将日志消息发送到调试输出窗口
AddEventLog	将日志消息发送到Windows事件日志，如果已部署到Windows Server，并希望将来自ASP.NET Core MVC应用程序的日志消息与其他类型的应用程序中的日志消息合并，这将非常有用

1．了解日志记录配置数据

AddConfiguration方法用于使用配置数据来配置日志记录系统，通常在appsettings.json文件中定义。代码清单14-37将名为Logging的配置部分添加到appsettings.json文件中，这部分配置对应于代码清单14-36中用于AddConfiguration方法的名称。

代码清单14-37　将配置部分添加到ConfiguringApps文件夹下

的appsettings.json文件中

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Logging配置部分指定应从不同的日志记录数据源显示的消息级别。日志记录系统支持7个级别的调试信息，表14-14按重要性对它们进行了排列。

表**14-14** 调试级别

调试级别	描述
Trace	用于在开发期间有用但在生产环境中不需要的消息
Debug	用于开发人员因调试问题所需的详细消息

Information	用于描述应用程序常规操作的消息
Warning	用于描述意外但不中断应用程序的事件的消息
Error	用于描述中断应用程序的错误的消息
Critical	用于描述灾难性故障的消息
None	用于禁用日志记录消息

代码清单14-37中的Default条目设置了将日志消息显示到Debug级别的阈值，也就是说，只显示Debug级别或更高级别的消息。其余条目将覆盖从特定命名空间记录消息的默认值，以便仅在Information级别或更高级别时才显示源自System或Microsoft命名空间的日志记录消息。要查看启用日志记录的效果，请通过选择Debug→Start Debugging，使用Visual Studio调试器启动应用程序。查看Output窗口，你将看到日志消息显示了如何处理每个HTTP请求，如下所示：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:65
417/
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActio
nInvoker[1]
      Executing action method ConfiguringApps.Controlle
rs.HomeController.Index
      (ConfiguringApps) with arguments (False) - ModelS
tate is Valid
info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.Vi
ewResultExecutor[1]
      Executing ViewResult, running view at path /Views
/Home/Index.cshtml.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActio
nInvoker[2]
      Executed action ConfiguringApps.Controllers.HomeC
ontroller.Index
      (ConfiguringApps) in 1597.3535ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 1695.6314ms 200 text/html; ch
arset=utf-8
```

2. 创建自定义日志消息

日志消息是由处理HTTP请求并生成响应的ASP.NET Core和MVC组件生成的。这种消息可以提供有用的信息，但也可以生成针对应用程序自定义日志消息，如代码清单14-38所示。

代码清单14-38　Controllers文件夹下的HomeController.cs文件中的自定义日志消息

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Logging;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;
        private ILogger<HomeController> logger;

        public HomeController(UptimeService up, ILogger
<HomeController> log) {
            uptime = up;
            logger = log;
        }

        public ViewResult Index(bool throwException = f
alse) {
            logger.LogDebug($"Handled {Request.Path} at
 uptime {uptime.Uptime}");

            if (throwException) {
                throw new System.NullReferenceException
();
            }
            return View(new Dictionary<string, string>
{
                ["Message"] = "This is the Index action
",
                ["Uptime"] = $"{uptime.Uptime}ms"
```

```
            });
        }

        public ViewResult Error() => View(nameof(Index)
,
            new Dictionary<string, string> {
                ["Message"] = "This is the Error action
"});
        }
}
```

ILogger接口定义了创建日志条目和获取实现该接口的对象所需的功能，HomeController类具有类型为ILogger的构造函数参数<HomeController>。类型参数允许日志记录系统在日志消息中使用类的名称，构造函数的参数值是通过依赖注入功能自动提供的，可参考第18章中的描述。

拥有ILogger后，可以使用Microsoft.Extensions.Logging命名空间中定义的扩展方法创建日志消息。表14-14描述了每种日志记录级别的方法。HomeController类使用LogDebug方法在Debug级别创建消息。要查看效果，请使用Visual

Studio调试器运行应用程序，并检查日志消息的Output窗口，其中的内容如下所示：

```
dbug: ConfiguringApps.Controllers.HomeController[0]
  Handled / at uptime 12
```

启动应用程序时会显示很多消息，很难挑选有用的消息。如果单击Output窗口顶部的Clear All按钮，然后重新加载浏览器，就更容易看到日志消息——这将确保仅显示与单个请求相关的日志消息。

14.5.4 配置依赖注入

ASP.NET Core应用程序的默认配置包括准备服务提供程序，第18章将详细介绍依赖项注入功能。代码清单14-39向Program类添加了配置语句。

代码清单14-39 在ConfiguringApps文件夹下的Program.cs文件中配置服务

```
using System;
```

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    config.AddJsonFile("appsettings.jso
n",
                        optional: true, reloadOnChange:
 true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logg
```

```
ing) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.Ge
tSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseDefaultServiceProvider((context, op
tions) => {
                    options.ValidateScopes =
                        context.HostingEnvironment.IsDe
velopment();
                })
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

UseDefaultServiceProvider方法使用内置的ASP.NET Core服务提供程序。还有一些其他可用的服务提供程序，但大多数项目可以使用内置功能，建议只有在遇到需要解决的特定问题时才使用第三方组件，并且需要对依赖注入有很好的理解，可参考第18章中的描述。

UseDefaultServiceProvider接收一个Lambda函数，这个Lambda函数接收WebHostBuilderContext对象和ServiceProviderOptions对象，用来配置内置的服务提供程序。唯一的配置属性名为ValidateScopes，在使用Entity Framework Core时需要禁用，如第8章所述。

14.6　配置MVC服务

在Startup类的ConfigureServices方法中调用AddMvc扩展方法时，会设置MVC应用程序所需的所有服务。这种方式非常方便，因为可以在一个步骤中注册所有MVC服务，但如果要改变默认的行为，那么确实需要做一些额外的工作来重新配置服务。

AddMvc扩展方法返回一个实现了IMvcBuilder接口的对象，MVC提供了一组可用于高级配置的扩

展方法，其中一些有用的扩展方法参见表14-15。另外，许多配置选项与后面章节中详细描述的功能有关。

表**14-15**　有用的扩展方法

扩展方法	描述
AddMvcOptions	用于配置MVC使用的服务
AddFormatterMappings	用于配置允许客户端指定接收的数据格式的功能
AddJsonOptions	用于配置JSON数据的创建方式
AddRazorOptions	用于配置Razor视图引擎
AddViewOptions	用于配置MVC处理视图的方式，包括使用哪些视图引擎

AddMvcOptions扩展方法用于配置最重要的MVC服务，它接收一个函数用以接收MvcOptions对象，该对象提供了一组配置属性，其中一些重要属性如表14-16所示。

表**14-16　MvcOptions**对象的一些重要属性

属性	描述
Conventions	返回模型约定的列表，这些约定用于自定义MVC如何创建控制器和操作
Filters	返回全局过滤器的列表
FormatterMappings	返回一个映射，以允许客户端指定接收的数据格式
InputFormatters	返回用于解析请求数据的对象列表
ModelValidatorProviders	返回用于验证数据的对象列表
OutputFormatters	返回类的列表，用于格式化从API控制器发送的数据
RespectBrowserAcceptHeader	指定在决定用于响应的数据格式时是否考虑Accept标头

这些配置属性用于微调MVC的运行方式，可以在指定的章节中找到与它们相关的功能的详细说明。但是，作为快速演示，代码清单14-40显示了如何使用AddMvcOptions方法更改配置选项。

代码清单14-40　在ConfiguringApps文件夹下更改Startup.cs文件中的配置选项

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<UptimeService>();
    services.AddMvc().AddMvcOptions(options => {
        options.RespectBrowserAcceptHeader = true;
    });
}
...
```

传递给AddMvcOptions方法的Lambda表达式接收一个MvcOptions对象，用它将RespectBrowserAcceptHeader属性设置为true。这使客户端对内容协商过程中选择的数据格式有了更强的控制，如第20章所述。

14.7　处理复杂配置

如果需要支持大量的托管环境，或者托管环境之间存在很多差异，那么在Startup类中使用if语句

进行分支配置可能会导致配置难以阅读且难以编辑，从而容易产生意外的变更。下面将介绍使用Startup类处理复杂配置的不同方法。

14.7.1　创建不同的外部配置文件

Program类执行的应用程序的默认配置会查找特定于运行应用程序的托管环境的JSON配置文件，因此可以使用名为appsettings.production.json的文件来存储特定于生产环境的配置。代码清单14-41恢复了将JSON文件加载到Program类的语句。

代码清单14-41　在ConfiguringApps文件夹下的Program.cs文件中加载托管环境文件

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }
        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    var env = hostingContext.HostingEnv
ironment;
                    config.AddJsonFile("appsettings.jso
n",
                            optional: true, reloadOnCha
nge: true)
                        .AddJsonFile($"appsettings.{env
.EnvironmentName}.json",
                            optional: true, reloadOnCha
nge: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logg
ing) => {
                    logging.AddConfiguration(
```

```
                        hostingContext.Configuration.Ge
tSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseDefaultServiceProvider((context, op
tions) => {
                    options.ValidateScopes =
                        context.HostingEnvironment.IsDe
velopment();
                })
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

从特定于平台的文件加载配置数据时，里面包含的配置设置将覆盖具有相同名称的任何现有数据。例如，使用ASP.NET Configuration File模板创建一个名为appsettings.development.json的文件，其中的配置数据如代码清单14-42所示。使用这个文件中的配置数据将EnableBrowserShortCircuit设置为false。

提　示

appsettings.development.json文件在创建后，好像会消失。在解决方案资源管理器中，如果在appsettings.json条目的左侧将箭头展开，你将看到Visual Studio对具有相似名称的项目进行了分组。

代码清单14-42　ConfiguringApps文件夹下的appsettings.development.json文件的内容

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": false
  }
}
```

appsettings.json文件将在应用程序启动时加载，如果应用程序在开发环境中运行，那么接着会加载appsettings.development.json文件。结果是，当应用程序在开发环境中运行时，EnableBrowserShortCircuit将为false，在暂存环境和生产环境中为true。

14.7.2　创建不同的配置方法

选择不同的配置数据文件可能很有用，但无法为复杂配置提供完整的解决方案，因为数据文件不包含C#语句。如果要更改用于创建服务或注册中间件的配置语句，则可以使用不同的方法，其中方法的名称包括托管环境，如代码清单14-43所示。

代码清单14-43　在Startup.cs文件中使用不同的方法名称

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Configuration;

namespace ConfiguringApps {
    public class Startup {

        public Startup(IConfiguration configuration) {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc().AddMvcOptions(options =>
{
                options.RespectBrowserAcceptHeader = tr
ue;
            });
        }

        public void ConfigureDevelopmentServices(IServi
ceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseExceptionHandler("/Home/Error");
            app.UseStaticFiles();
```

```
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }

        public void ConfigureDevelopment(IApplicationBu
ilder app,
                IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseBrowserLink();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

当ASP.NET Core在Startup类中查找ConfigureServices和Configure方法时，首先检查是否存在包含托管环境名称的方法。以上代码添加了ConfigureDevelopmentServices方法，用于代替开发环境中的ConfigureServices方法；还添加了ConfigureDevelopment方法，用于代替开发环境中的Configure方法。可以为需要支持的每个环境定义

单独的方法，如果没有可用的特定于环境的方法，则依赖要调用的默认方法。在该例中，ConfigureServices和Configure方法将用于暂存环境与生产环境。

警　告

如果定义了特定于环境的方法，则不会调用默认方法。例如，在代码清单14-43中，ASP.NET Core不会在开发环境中调用Configure方法，因为有ConfigureDevelopment方法。这意味着每个方法都负责环境所需的完整配置。

14.7.3 创建不同的配置类

使用不同的方法意味着不必使用if语句来检查托管环境的名称，但可能导致类变得很大，这本身就是问题。对于特别复杂的配置，最后的措施是为每个托管环境创建不同的配置类。当ASP.NET查找Startup类时，首先检查是否存在名称包含当前托管环境的类。为此，在项目中添加一个名为StartupDevelopment.cs的类文件，并用它定义代码清单14-44所示的类。

代码清单14-44 ConfiguringApps文件夹下的StartupDevelopment.cs文件的内容

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class StartupDevelopment {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UptimeService>();
```

```
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseBrowserLink();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

这个类包含特定于开发托管环境的ConfigureServices和Configure方法。要使ASP.NET Core能够找到特定于环境的Startup类，需要对Program类进行更改，如代码清单14-45所示。

代码清单14-45　在Program.cs文件中启用特定于环境的Startup

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDir
ectory())
                .ConfigureAppConfiguration((hostingCont
ext, config) => {
                    var env = hostingContext.HostingEnv
ironment;
                    config.AddJsonFile("appsettings.jso
n",
                        optional: true, reloadOnChange:
 true)
                    .AddJsonFile($"appsettings.{env.Env
ironmentName}.json",
                            optional: true, reloadOnCha
nge: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logg
ing) => {
```

```
                    logging.AddConfiguration(
                        hostingContext.Configuration.Ge
tSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseDefaultServiceProvider((context, op
tions) => {
                    options.ValidateScopes =
                        context.HostingEnvironment.IsDe
velopment();
                })
                .UseStartup(nameof(ConfiguringApps))
                .Build();
        }
    }
}
```

UseStartup方法不指定类，而是给出应该使用的程序集的名称。当应用程序启动时，ASP.NET将查找名称中包含托管环境的类，例如StartupDevelopment或StartupProduction。如果不存在，就返回使用常规的Startup类。

14.8　小结

本章解释了如何配置MVC应用程序，还描述了Program和Startup类的作用以及它们提供的默认配置选项，展示了如何使用管道处理请求以及如何使用不同类型的中间件来控制请求流和它们引发的响应。下一章将介绍路由系统，以便MVC将请求URL映射到控制器和操作方法。

第15章　URL路由

早期版本的ASP.NET假设请求的URL与服务器硬盘上的文件之间存在直接关系。服务器的工作是从浏览器接收请求并从相应的文件传递输出。这种方法适用于Web窗体，其中每个ASPX页面都是文件和对请求的自包含响应。

这对MVC应用程序来说并无意义，请求由控制器类中的操作方法处理，而且与硬盘上的文件也没有一对一关联。

为了处理MVC的URL，ASP.NET平台使用了路由系统，路由系统已针对ASP.NET Core进行了大量修改。本章将展示如何使用路由系统为项目创建强大而灵活的URL处理方式。正如你将看到的，路由系统允许你创建所需的任何URL模式，并以清晰

简洁的方式表达它们。路由系统有两个功能。

- 检查传入的URL并选择控制器和操作方法来处理请求。
- 生成传出的URL。这些是在视图呈现的HTML中显示的URL，以便在用户单击链接时调用特定的操作（此时将再次成为传入的URL）。

本章将重点介绍路由并使用它们来处理传入的URL，以便用户可以访问控制器和操作。在MVC应用程序中创建路由有两种方法——基于约定的路由和属性路由。本章将解释这两种方法。

然后，下一章将展示如何使用这些路由生成包含在视图中的传出URL，以及如何自定义路由系统并使用称为area（区域）的相关功能。表15-1总结了路由的相关问题。

表**15-1**　路由的相关问题

问题	答案
路由是什么？	路由系统负责处理传入的请求并选择控制器和操作方法来处理它们。路由系统还用于在视图中生成路由，称为传出的URL
路由有什么用？	路由系统能够灵活地处理请求，而不是将URL与Visual Studio项目中的类结构关联到一起
如何使用路由？	URL与控制器和操作方法之间的映射定义在Startup.cs文件中，也可通过将Route属性应用于控制器来实现
有什么缺点或限制吗？	复杂应用程序的路由配置可能会变得难以管理
有其他代替方式吗？	没有。路由系统是ASP.NET Core的组成部分

表15-2列出了本章要介绍的操作。

表**15-2**　本章要介绍的操作

操作	方法	代码清单
创建URL和操作方法之间的映射	创建路由	代码清单15-9
省略URL片段	为路由片段定义默认值	代码清单15-10～代码清单15-12

匹配没有相应路由变量的URL片段	定义静态片段	代码清单15-13～代码清单15-16
将URL片段传递给操作方法	自定义片段变量	代码清单15-17～代码清单15-19
省略没有默认值的URL片段	定义可选片段	代码清单15-20和代码清单15-21
定义路由来匹配任意数量的URL片段	使用catchall片段	代码清单15-22和代码清单15-23
限制路由可以匹配的URL	应用路由约束	代码清单15-24～代码清单15-33
在控制器中定义路由	使用属性路由	代码清单15-34～代码清单15-38

15.1　准备示例项目

本章将使用ASP.NET Core Web Application（.NET Core）模板创建名为UrlsAndRoutes的Empty项目。为了添加对MVC框架、开发人员错误页面和静态文件的支持，将代码

清单15-1显示的语句添加到Startup类中。

代码清单15-1　在UrlsAndRoutes文件夹下的Startup.cs文件中配置应用程序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc();
        }
    }
}
```

15.1.1 创建模型类

本章中的所有工作都是为了将请求URL与操作相匹配。需要的唯一模型类会传递有关处理请求的控制器和操作方法的详细信息。创建Models文件夹并添加一个名为Result.cs的类文件，用来定义代码清单15-2所示的类。

代码清单15-2 Models文件夹下的Result.cs文件的内容

```
using System.Collections.Generic;

namespace UrlsAndRoutes.Models {

    public class Result {
        public string Controller { get; set; }

        public string Action { get; set; }

        public IDictionary<string, object> Data { get; set; }
            = new Dictionary<string, object>();
    }
}
```

Controller和Action属性将用于指示请求的处理

方式，Data字典将用于存储路由系统生成的请求的其他详细信息。

15.1.2　创建Example控制器

下面使用一些简单的控制器来演示路由的工作原理。创建Controllers文件夹并添加一个名为HomeController.cs的类文件，其中的内容如代码清单15-3所示。

代码清单15-3　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });
    }
```

```
}
```

使用Home控制器定义的Index操作方法调用View方法来渲染名为Result的视图（在15.1.3节中定义），并提供Result对象作为模型对象。模型对象的属性使用nameof函数设置，用来指示已使用哪个控制器和操作方法来处理请求。

按照相同的方式，将CustomerController.cs文件添加到Controllers文件夹中，并用它定义代码清单15-4所示的Customer控制器。

代码清单15-4　Controllers文件夹下的CustomerController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
```

```
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });

        public ViewResult List() => View("Result",
            new Result {
                Controller = nameof(CustomerController)
,
                Action = nameof(List)
            });
    }
}
```

最后一个控制器定义在名为AdminController.cs的文件中，将该文件添加到Controllers文件夹中，文件的内容如代码清单15-5所示。

代码清单15-5　Controllers文件夹下的AdminController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class AdminController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
```

```
                Controller = nameof(AdminController),
                Action = nameof(Index)
            });
    }
}
```

15.1.3　创建视图

我们在15.1.2节定义的所有操作方法中指定了Result视图，从而允许创建一个将由所有控制器共享的视图。创建Views/Shared文件夹，并添加一个名为Result.cshtml的新视图，内容如代码清单15-6所示。

代码清单15-6　Views/Shared文件夹下的Result.cshtml文件的内容

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
</body>
</html>
```

Result视图包含一个表格，其中显示了使用Bootstrap设置样式的模型对象的属性。为了向项目添加Bootstrap，使用Bower Configuration File模板创建bower.json文件，并将Bootstrap包添加到dependencies部分，如代码清单15-7所示。

代码清单15-7　在UrlsAndRoutes文件夹下的bower.json文件中添加Bootstrap包

```
{
  "name": "asp.net",
```

```
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

最后的准备工作是在Views文件夹中创建_ViewImports.cshtml文件，以设置用于Razor视图的内置标签助手，并导入Modes命名空间，如代码清单15-8所示。

代码清单15-8　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using UrlsAndRoutes.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Startup类中的配置不包含任何有关MVC如何将HTTP请求映射到控制器和操作的说明。当运行应用程序时，请求的任何URL都将得到404-Not Found响应，如图15-1所示。

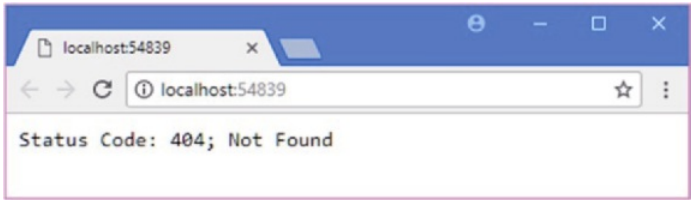


图15-1　运行示例应用程序的响应

15.2　介绍URL模式

路由系统使用一组路由来工作。这些路由共同包含应用程序的URL架构或方案，这是应用程序将识别和响应的一组URL。

不需要手动输入应用程序中支持的所有单个URL。相反，每个路由都包含一种URL模式，用来与传入的URL进行比较。如果传入的URL与模式匹配，路由系统将使用匹配的模式来处理URL。下面是一个简单的示例URL：

```
http://mysite.com/Admin/Index
```

网址可以细分为片段。这些是URL的组成部

分，不包括由/字符分隔的主机名和查询字符串。在上面的示例URL中，有两个片段，如图15-2所示。

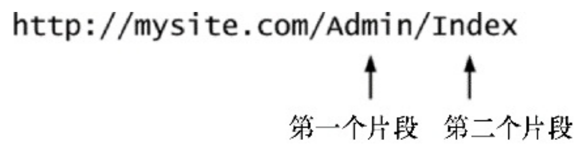


图15-2　示例URL中的片段

第一个片段包含单词Admin，第二个片段包含单词Index。显然，第一个片段与控制器有关，而第二个片段涉及操作。但是，需要使用路由系统可以理解的URL模式来表达这种关系。以下是与示例URL匹配的网址格式：

```
{controller}/{action}
```

当处理传入的HTTP请求时，路由系统的功能是将请求的URL与模式匹配，并从URL中提取模式中定义的片段变量的值。

片段变量使用大括号括起来。示例模式具有两个名为controller和action的片段变量，因此controller片段变量的值将为Admin，而action片段变量的值将为Index。

MVC应用程序通常会有多个路由，路由系统会将传入的URL与每个路由的URL模式进行比较，直到找到匹配为止。默认情况下，一个模式将匹配具有正确片段数量的任何URL。例如，模式{controller}/{action}将匹配具有两个片段的任何URL，如表15-3所示。

表**15-3**　匹配的**URL**

请求的**URL**	片段变量
*****//mysite.***/Admin/Index	controller片段变量的值为Admin，action片段变量的值为Index
*****//mysite.***/Admin	没有匹配，片段太少

*****//mysite.***/Admin/Index/Soccer	没有匹配，片段太多

表15-3突出显示了URL模式的两个关键行为。

- URL模式匹配的片段数是固定的。它们将仅匹配与模式具有相同数量片段的URL。表15-3的第2个和第3个示例展示了这一点。
- URL模式匹配的片段的内容是不固定的。如果URL具有正确的片段数，URL模式将提取每个片段变量的值，无论它是什么。

这些默认行为是了解URL模式如何运作的关键。本章后面将介绍如何更改默认值。

15.3　创建和注册简单路由

一旦有了URL模式，就可以用它定义路由。路由定义在Startup.cs文件中，并作为参数传递给UseMvc方法，该方法用于在Configure方法中设置MVC。代码清单15-9显示了一个基本路由，用于将

请求映射到示例应用程序中的控制器。

代码清单15-9　在UrlsAndRoutes文件夹下的Startup.cs文件中定义基本路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "default", templa
te: "{controller}/{action}");
            });
        }
    }
```

```
}
```

路由是使用Lambda表达式创建的，Lambda表达式则作为参数传递给UseMvc配置方法。Lambda表达式接收一个对象，它实现了Microsoft.AspNetCore.Routing命名空间中的IRouteBuilder接口，并使用MapRoute扩展方法定义路由。为了使路由更容易理解，约定在调用MapRoute扩展方法时指定参数名称，这就是在代码中明确命名name和template参数的原因。name参数指定了路由的名称，template参数用于定义模式。

提　示

命名路由是可选的，但有观点认为，这样无法

做到彻底的关注点分离。

可以通过启动示例应用程序来查看这里对路由所做更改的效果。应用程序首次启动时没有变化——你仍会看到404错误，但如果导航到与{controller}/{action}模式匹配的URL，你将看到图15-3所示的结果，这表明已导航到/Admin/Index。

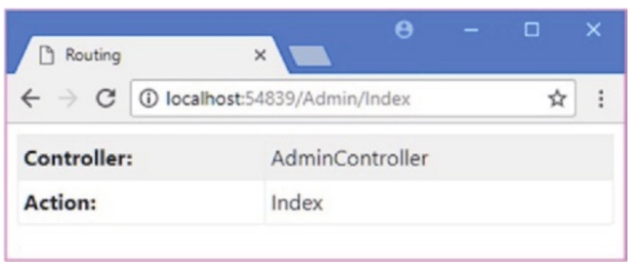


图15-3　使用简单的路由进行导航

应用程序的根URL不起作用，原因是添加到Startup.cs文件的路由没有告诉MVC，在请求的URL没有片段时如何选择控制器类和操作方法。15.4节将介绍如何解决这个问题。

15.4　定义默认值

请求默认URL时，示例应用程序返回404错误，因为默认URL与Startup类中定义的路由模式不匹配。由于默认URL中没有可以与路由模式定义的controller和action变量匹配的片段，因此无法匹配路由系统。

之前解释过，URL模式只匹配具有指定片段数的URL。更改此行为的一种方法是使用默认值。当URL不包含可由路由模式匹配的片段时，将应用默认值。代码清单15-10定义了一个使用默认值的路由。

代码清单15-10　在UrlsAndRoutes文件夹下的Startup.cs文件中提供默认值

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller}/{action}",
                    defaults: new { action = "Index" });

            });
        }
    }
}
```

默认值可作为匿名类型的属性提供，并作为defaults参数传递给MapRoute方法。在以上代码中，为action变量提供了默认值Index。

这个路由将匹配所有两段式URL，就像以前一样。例如，如果请求******//mydomain.***/Home/Index，路由将提取Home作为controller变量的值，并将提取Index作为action变量的值。

但是现在action变量有了默认值，路由也将匹配单片段URL。当处理单片段URL时，路由系统将从URL中提取controller变量的值，并使用action变量的默认值。通过这种方式，用户在请求/Home的时候，MVC将在Home控制器上调用Index操作方法，如图15-4所示。

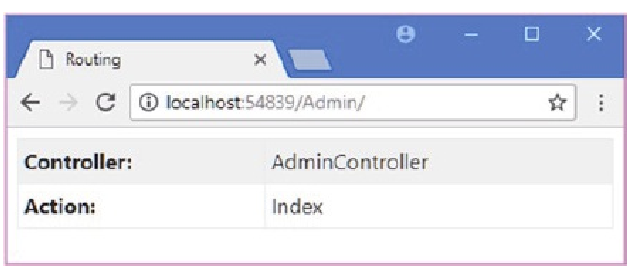


图15-4　使用默认的操作方法

定义内联默认值

默认值也可以表示为URL模式的一部分，这是一种更简洁的表达路由的方式，如代码清单15-11所示。内联语法只能为URL模式的部分变量提供默认值，但是，正如你将了解到的那样，能够在这种模式之外提供默认值通常很有用。因此，了解表达默认值的两种方法是很有用的。

代码清单15-11　在UrlsAndRoutes文件夹下的Startup.cs文件中定义内联默认值

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
```

```
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller}/{action=Ind
ex}");
            });
        }
    }
}
```

可以进一步匹配完全不包含任何片段变量的URL，仅依靠默认值来标识action和controller。作为示例，代码清单15-12显示了如何通过为两个片段提供默认值来映射应用程序的根URL。

代码清单15-12　在UrlsAndRoutes文件夹下的Startup.cs文件中提供默认值

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                template: "{controller=Home}/{action=In
dex}");
            });
        }
    }
}
```

通过为controller和action变量提供默认值，路由将匹配具有零个、一个或两个片段的URL，如表15-4所示。

表**15-4**　匹配**URL**

片段数	示例	匹配结果
0	/	controller = Home, action = Index
1	**/Customer**	controller = Customer, action = Index
2	**/Customer/List**	controller = Customer, action = List
3	**/Customer/List/All**	没有匹配，片段太多

传入的URL中接收的片段越少，路由越依赖于默认值，直到仅使用默认值匹配没有片段的URL。

可以通过启动示例应用程序来查看默认值的效果。当浏览器请求应用程序的根URL时，将使用controller和action片段变量的默认值，这将导致MVC在Home控制器上调用Index操作方法，如图15-5所示。

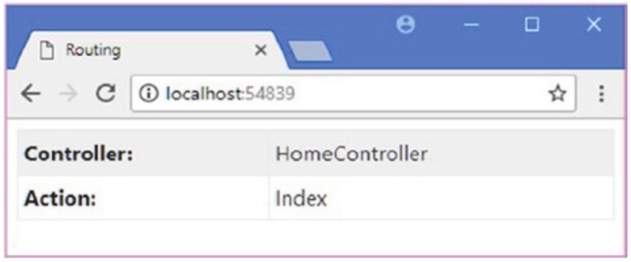


图15-5　使用默认值扩大路由范围

15.5　使用静态URL片段

并非URL格式中的所有片段都必须是变量。还可以创建具有静态片段的模式。假设应用程序需要匹配以Public为前缀的URL，如下所示：

```
****://mydomain.***/Public/Home/Index
```

这可以通过使用代码清单15-13所示的URL模式来完成。

代码清单15-13　在UrlsAndRoutes文件夹下的Startup.cs文件中使用静态片段

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}");

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}
/{action=Index}");
            });
        }
    }
}
```

这种新模式将仅匹配包含3个片段的URL，第一个片段必须为Public。其他两个片段可以包含任何值，并将用于controller和action变量。如果省略最后两个片段，将使用默认值。

还可以创建包含静态和可变元素的片段的URL模式，如代码清单15-14所示。

代码清单15-14　UrlsAndRoutes文件夹下的Startup.cs文件中的混合片段

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute("", "X{controller}/{act
ion}");

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}");

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}
/{action=Index}");
            });
        }
    }
}
```

这个路由中的模式能匹配任何两片段的URL，其中第一个片段以字母X开头。controller的值取自第一个片段，不包括X。action的值取自第二个片段。如果启动应用程序并导航到/XHome/Index，路由效果如图15-6所示。

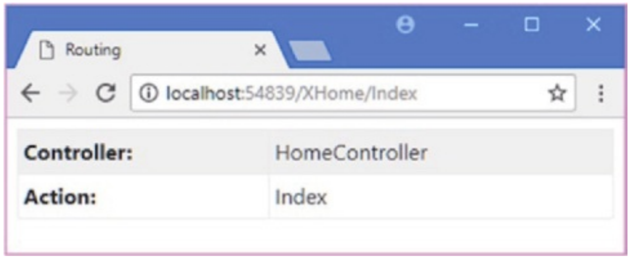


图15-6　在单个片段中混合静态和可变元素的路由效果

路由排序

代码清单15-14定义了一个新的路由并将其放在所有其他路由之前。这样做是为了让路由按照定义的顺序应用。MapRoute方法将路由添加到路由配置的末尾，这意味着路由通常会按定义的顺序应用。之所以说“通常”，是因为有一些方法还可以在特定位置插入路由。不建议使用这些方法，因为按照定义的顺序应用路由能够使应用程序的路由更容易理解。

路由系统尝试对传入的URL与首先定义的路由

的URL模式进行匹配，只有在没有匹配时才进入下一个路由。这些路由按顺序尝试，直到找到匹配的路由或路由被用完。因此，必须首先定义最具体的路由。代码清单15-14添加的路由比后面的路由更具体。假设颠倒路由的顺序，如下所示：

```
...
routes.MapRoute("MyRoute", "{controller=Home}/{action=Index}");
routes.MapRoute("", "X{controller}/{action}");
...
```

然后，第一个路由能够匹配任何具有零个、一个或两个片段的URL，因而将始终使用。更具体的路由，比如上述代码中的第二个路由，则永远不会使用。新路由排除了URL的前导X，但旧路由不会这样做。因此，像下面这样的URL：

```
http://mydomain.com/XHome/Index
```

将被定位到名为XHome的控制器，我们假设应

用程序中有XHomeController类，并且有名为Index的操作方法。

你可以组合静态URL片段和默认值以创建特定URL的别名。使用的URL模式在部署应用程序时与用户形成了约定，如果随后重构应用程序，则需要保留以前的URL模式，以便用户创建的任何URL收藏夹、宏或脚本能够继续工作。

想象一下，以前有一个名为Shop的控制器，现在已被Home控制器取代了。代码清单15-15显示了如何创建保留旧URL模式的路由。

代码清单15-15　UrlsAndRoutes文件夹下的Startup.cs文件中的片段和默认值

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "ShopSchema",
                    template: "Shop/{action}",
                    defaults: new { controller = "Home"
 });

                routes.MapRoute("", "X{controller}/{act
ion}");

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}");

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}
```

```
/{action=Index}");
            });
        }
    }
}
```

以上加粗显示的路由能匹配任何两个片段的URL，其中第一个片段是Shop。action的值取自第二个URL片段。URL模式不包含controller的变量片段，因此使用默认值。defaults参数提供了controller的值，因为没有片段能够将值作为URL模式的一部分应用于该URL模式。

其结果是，Shop控制器上的操作请求被转换为对Home控制器的请求。可以通过启动应用程序并导航到/Shop/Index来查看路由效果。如图15-7所示，新路由使MVC在Home控制器中定位到Index操作方法。

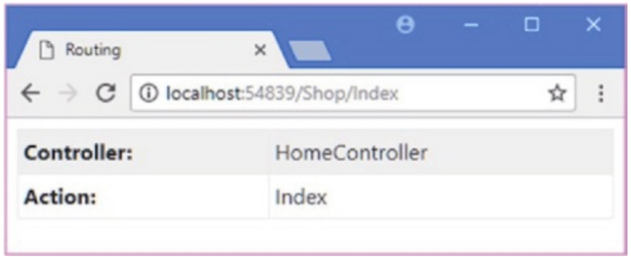


图15-7　创建别名以保留URL模式

可以更进一步，为已经重构并且控制器中已不存在的操作方法创建别名。为此，创建一个静态URL，并提供controller和action的默认值，如代码清单15-16所示。

代码清单15-16　在UrlsAndRoutes文件夹下的Startup.cs文件中为controller和action使用别名

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: "ShopSchema2",
                    template: "Shop/OldAction",
                    defaults: new { controller = "Home"
, action = "Index" });
                routes.MapRoute(
                    name: "ShopSchema",
                    template: "Shop/{action}",
                    defaults: new { controller = "Home"
 });

                routes.MapRoute("", "X{controller}/{act
ion}");

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}");

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}
/{action=Index}");
            });
        }
```

```
    }
}
```

请注意，首先要定义新路由，因为它比后面的路由更具体。例如，如果对Shop/OldAction的请求由下一个定义的路由处理，并且有一个带有OldAction操作方法的控制器，则可能无法得到想要的结果。

15.6　定义自定义片段变量

controller和action片段变量在MVC应用程序中具有特殊含义，相当于为请求提供服务的控制器和操作方法。这些只是内置的片段变量，也可以定义自定义片段变量，如代码清单 15-17所示（这里已经删除了已有路由，以便重新开始）。

代码清单15-17　在UrlsAndRoutes文件夹下的Startup.cs文件中定义其他变量

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller=Home}/{actio
n=Index}/{id=DefaultId}");
            });
        }
    }
}
```

以上代码中的URL模式定义了标准的controller和action变量，以及名为id的自定义变量。路由将匹

配0～3个片段的URL。第3个片段的内容将分配给id变量，如果没有第3个片段，就使用默认值。

警　告

某些名称是保留的，不适合作为自定义片段变量的名称，它们是controller、action、area和page，前两个名称的含义是显而易见的。area、page由Razor页面的使用，本书将在下一章解释。

Controller类是控制器的基础，它定义了RouteData属性，该属性返回一个Microsoft.AspNetCore.Routing. RouteData对象，该

对象提供有关路由系统的详细信息以及当前请求的路由方式。在控制器中，可以使用RouteData.Values属性访问操作方法中的任何片段变量，该属性会返回一个包含片段变量的字典。为了演示，这里在Home控制器中添加一个名为CustomVariable的操作方法，如代码清单15-18所示。

代码清单15-18　在Controllers文件夹下的HomeController.cs文件中访问片段变量

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable() {
            Result r = new Result {
```

```
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = RouteData.Values["id"];
            return View("Result", r);
        }
    }
}
```

操作方法CustomVariable使用RouteData.Values属性获取路由URL模式中的自定义片段变量id的值，RouteData.Values属性则返回路由系统生成的变量字典。自定义片段变量将被添加到视图模型对象中，可以通过运行应用程序并请求以下URL来查看：

```
/Home/CustomVariable/Hello
```

路由模板对以上URL中的第3个片段与id变量的值进行匹配，产生图15-8所示的结果。

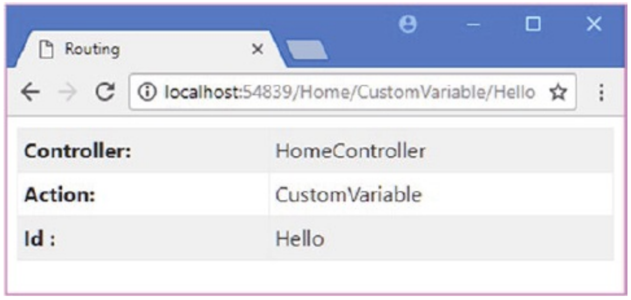


图15-8　显示自定义片段变量的值

代码清单15-17中的URL模式定义了id变量的默认值，这意味着路由也可以匹配具有两个片段的URL。可以通过请求如下URL来查看如何使用默认值：

```
/Home/CustomVariable
```

路由系统使用自定义片段变量的默认值，如图15-9所示。

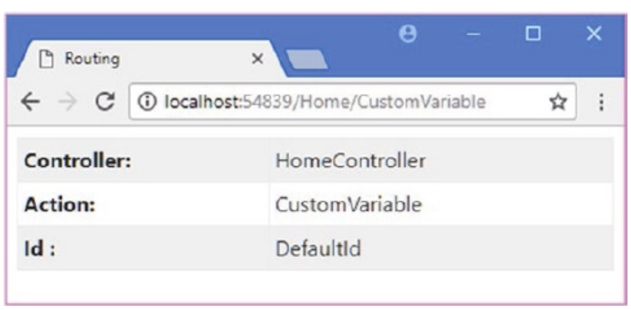


图15-9　自定义片段变量的默认值

15.6.1　使用自定义片段变量作为操作方法的参数

使用RouteData.Values集合只是访问自定义片段变量的一种方式，另一种方式可能更优雅。如果操作方法定义了名称与URL模式变量匹配的参数，MVC将把自动从URL获取的值作为参数传递给操作方法。

在代码清单15-17所示的路由中定义的自定义片段变量名为id。这里可以修改Home控制器中的CustomVariable操作方法，使其具有相同名称的参数，如代码清单15-19所示。

代码清单15-19　在Controllers文件夹下的HomeController.cs文件中添加action参数

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;
```

```
namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id;
            return View("Result", r);
        }
    }
}
```

当路由系统对URL与代码清单15-17中定义的路由进行匹配时，URL中第3个片段的值将被分配给自定义片段变量 id。MVC 对片段变量的列表与操作方法的参数列表进行比较，如果名称匹配，就将值从URL传递给操作方法。

id参数的类型是字符串，但MVC会尝试将URL值转换为使用的任何参数类型。如果操作方法将id参数声明为int或DateTime，那么MVC将从URL中接收值并转换为相应类型。这是一个优雅而实用的功能，无须自己处理转换。可以通过启动应用程序并请求/Home/CustomVariable/Hello来查看操作方法参数的效果，从而生成图15-10所示的结果。如果省略第3个片段，将为操作方法提供默认片段值。

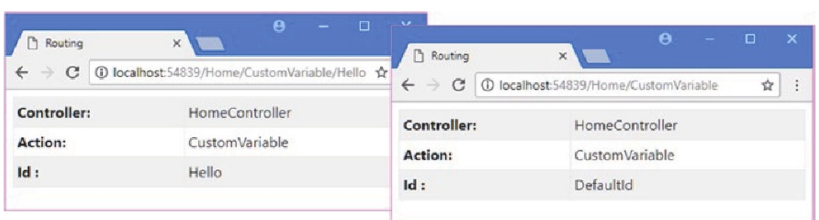


图15-10　使用操作方法参数访问片段变量的结果

注　意

MVC使用模型绑定功能将URL中包含的值转换为.NET类型，模型绑定还可以处理比此例更复杂的情况。第26章将介绍模型绑定。

15.6.2 定义可选的URL片段

可选的URL片段是用户不需要指定的URL片段，而且没有指定默认值。可选片段由片段名称后面的问号（?字符）表示，如代码清单15-20所示。

代码清单15-20 在UrlsAndRoutes文件夹下的Startup.cs文件中指定可选片段

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
            routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=In
dex}/{id?}");
            });
        }
    }
}
```

无论是否已提供id片段，以上路由都将匹配URL。表15-5显示了可选片段如何适用于不同的URL。

表**15-5**　使用可选片段变量匹配**URL**

片段编号	示例**URL**	匹配结果

0	/	controller = Home, action = Index
1	/Customer	controller = Customer, action = Index
2	/Customer/List	controller = Customer, action = List
3	/Customer/List/All	controller = Customer, action = List id = All
4	/Customer/List/All/Delete	没有匹配，片段太多

从表15-5可以看出，只有当传入的URL中存在相应的片段时，才会将id变量添加到变量集合中。如果需要知道用户是否为片段变量提供了值，这个功能将非常有用。如果没有为可选片段变量提供值，相应参数的值将为null。在代码清单15-21中，因为没有为id片段变量提供值，所以更新了Home控制器以进行响应。

代码清单15-21　在Controllers文件夹下的HomeController.cs文件中检查片段

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;
```

```
namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id ?? "<no value>";
            return View("Result", r);
        }
    }
}
```

图15-11显示了启动应用程序并导航到/Home/CustomVariable URL的结果，这个URL不包含id片段变量的值。

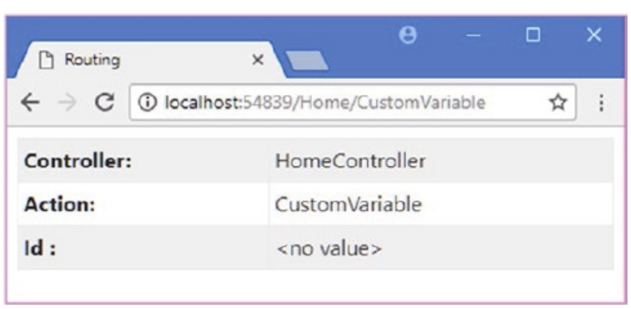


图15-11　检测URL何时不包含可选片段变量的值

理解默认的路由配置

将添加MVC到Startup类时，可以使用UseMvcWithDefaultRoute方法执行此操作。这只是设置最常见路由配置的便捷方法，相当于以下代码：

```
...
app.UseMvc(routes => {
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id
?}");
});
...
```

默认配置匹配按名称定位控制器类和操作方法的URL，并带有可选的id片段。如果控制器或操作片段丢失，则默认值分别用于定位Home控制器和

Index操作方法。

15.6.3　定义可变长度路由

更改URL模式的默认保守性的另一种方法是接收可变数量的URL片段，这允许你在单个路由中匹配任意长度的URL。可以通过将其中一个片段变量指定为catchall来定义对变量片段的支持，可通过在其前面添加星号（*字符）来完成，如代码清单15-22所示。

代码清单15-22　在UrlsAndRoutes文件夹下的Startup.cs文件中指定catchall变量

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {

    public class Startup {
        public void ConfigureServices(IServiceCollectio
```

```
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=In
dex}/{id?}/{*catchall}");
            });
        }
    }
}
```

这里通过扩展前一个例子中的路由来添加catchall片段变量，catchall是一个十分富有想象力的名称。这个路由现在将匹配任何URL，而无论其包含多少片段以及这些片段的值如何。前3个片段分别用于设置controller、action和id变量的值。如果URL包含其他片段，则把它们都分配给catchall变量，如表15-6所示。

表**15-6**　使用**catchall**变量匹配**URL**

片段编号	示例**URL**	匹配结果
0	/	controller = Home, action = Index
1	/Customer	controller = Customer, action = Index
2	/Customer/List	controller = Customer, action = List
3	/Customer/List/All	controller = Customer, action = List id = All
4	/Customer/List/All/Delete	controller = Customer, action = List id = All catchall = Delete
5	/Customer/List/All/Delete/Perm	controller = Customer, action = List id = All catchall = Delete/Perm

代码清单15-23更新了Customer控制器，以便List操作通过模型对象将catchall变量的值传递给视图。

代码清单15-23　更新Controllers文件夹下的CustomerController.cs文件中的操作方法

```
using Microsoft.AspNetCore.Mvc;
```

```
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });
        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["Id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catc
hall"];
            return View("Result", r);
        }
    }
}
```

要测试catchall片段，请运行应用程序并请求以下URL：

```
/Customer/List/Hello/1/2/3
```

以上路由中的URL模式将匹配的片段数量没有

上限。图15-12显示了catchall片段的效果。请注意，catchall捕获的片段以segment/segment/segment的形式显示，并且需要处理字符串以分解各个片段。

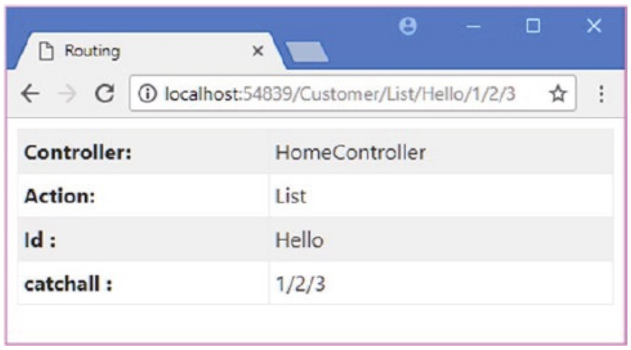


图15-12　使用catchall片段

15.7　约束路由

URL模式在匹配URL中的片段数时是保守的，当它们与片段的内容匹配时是自由的。前几节已经解释了控制保守程度的不同技术——使用默认值、可选变量等使路由匹配更多或更少的片段。

现在来看看如何控制匹配URL片段内容的自由

度，也就是如何限制路由将匹配的URL集合。代码清单15-24演示了如何使用简单约束来限制路由匹配的URL。

代码清单15-24　在UrlsAndRoutes文件夹下的Startup.cs文件中约束路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
```

```
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id:int?}");
            });
        }
    }
}
```

使用冒号（：字符）将约束与片段变量名分开。以上代码中的约束是int，已应用于id片段。这是一个内联约束，且被定义为URL模式的一部分，应用于单个片段：

```
...
template: "{controller}/{action}/{id:int?}",
...
```

int约束仅允许URL模式匹配值可以解析为整数值的片段。id片段是可选的，因此路由将匹配省略了id的片段，但如果片段存在，那么id必须是整数值，如表15-7所示。

表**15-7**　使用约束匹配**URL**

示例**URL**	匹配结果
/	controller = Home, action = Index, id = null
/Home/CustomVariable/Hello	没有匹配，id片段无法被解析为int值
/Home/CustomVariable/1	controller = Home, action = CustomVariable, id = 1
/Home/CustomVariable/1/2	没有匹配，片段太多

在定义路由时，也可以使用MapRoute方法的constraints参数在URL模式之外指定约束。如果希望将URL模式与其约束分开，或者更喜欢早期版本的MVC（不支持内联约束）所使用的路由样式，那么这种方法很有用。代码清单15-25显示了相同的id片段变量的整数约束，可使用单独的约束来表示。使用这种格式时，默认值也在外部表示。

代码清单15-25　在UrlsAndRoutes文件夹下的Startup.cs文件中表达约束

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller}/{action}/{id?}",

                defaults: new { controller = "Home", ac
tion = "Index" },
                constraints: new { id = new IntRouteCon
straint() });
            });
        }
    }
}
```

MapRoute方法的constraints参数是使用匿名类型定义的，匿名类型的属性名称对应于受约束的片段变量。Microsoft.AspNetCore.Routing.Constraints命名空间包含一组可用于定义各个约束的类。在代码清单15-25中，constraints参数被配置为对id片段使用IntRouteConstraint对象，以创建与代码清单15-24所示的内联约束相同的效果。

表15-8描述了Microsoft.AspNetCore.Routing命名空间中完整的约束类。Constraints命名空间及等效的内联约束可以应用于URL模式中的单个片段，后面的章节将描述其中一些情况。

提　示

可以使用MVC提供的一组特性（例如HttpGet和HttpPost特性）对使用特定HTTP谓词（例如GET或POST）请求的操作方法进行限制。有关使用这些特性处理控制器中表单的详细信息，请参阅第7章，有关此类可用特性的完整列表，请参阅第20章。

表15-8　片段级路由约束

内联约束	描述	类名
alpha	匹配字母字符，无论大小写（A～Z、a～z）	AlphaRouteConstraint()
bool	匹配可以解析为bool类型的值	BoolRouteConstraint()
datetime	匹配可以解析为DateTime类型的值	DateTimeRouteConstraint()
decimal	匹配可以解析为decimal类型的值	DecimalRouteConstraint()

double	匹配可以解析为double类型的值	DoubleRouteConstraint()
float	匹配可以解析为float类型的值	FloatRouteConstraint()
guid	匹配GUID值	GuidRouteConstraint()
int	匹配可以解析为int类型的值	IntRouteConstraint()
length(len)/length(min,max)	匹配具有指定字符数量的值，或者匹配字符数量介于最小和最大字符数量之间的值	LengthRouteConstraint(len) LengthRouteConstraint(min, max)
long	匹配可以解析为long类型的值	LongRouteConstraint()
maxlength(len)	匹配字符数量不超过len的字符串	MaxLengthRouteConstraint(len)
max(val)	配置一个int值，该值小于val	MaxRouteConstraint(val)
minlength(len)	匹配字符数量至少为len的字符串	MinLengthRouteConstraint(len)
min(val)	匹配一个int值，该值大于val	MinRouteConstraint(val)
range(min,max)	匹配一个int值，该值介于min和max之间	RangeRouteConstraint(min, max)

regex(expr)	匹配正则表达式	RegexRouteConstraint(expr)

15.7.1　使用正则表达式约束路由

正则表达式提供了灵活性最大的约束，可使用正则表达式匹配片段。代码清单15-26对controller片段进行了约束以限制匹配的URL范围。

代码清单15-26　在UrlsAndRoutes文件夹下的Startup.cs文件中使用正则表达式

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller:regex(^H.*)=Home
}/{action=Index}/{id?}");
            });
        }
    }
}
```

以上代码使用约束来限制路由，使路由只匹配controller片段以字母H开头的URL。

注　意

在检查约束之前会应用默认值。因此，举例来说，如果请求URL /，就应用controller的默认值

Home。然后检查约束，并且由于controller的值以H开头，因此默认URL将匹配路由。

正则表达式可以约束路由，以便只有URL片段的特定值才能匹配。这是使用竖线（|）字符完成的，如代码清单15-27所示（这里将URL模式拆分成两行以适应纸面宽度，在实际项目中不用这么做）。

代码清单15-27　在UrlsAndRoutes文件夹下的Startup.cs文件中约束路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
```

```
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                  template: "{controller:regex(^H.*)=Ho
me}/"
                    + "{action:regex(^Index$|^About$)=I
ndex}/{id?}");
            });
        }
    }
}
```

以上约束允许路由仅匹配action片段的值为Index或About的URL。由于约束将被一起应用，因此需要组合action变量与controller变量的限制。这意味着仅当controller变量以字母H开头且action变量为Index或About时，代码清单15-27中的路由才能匹配URL。

15.7.2　使用类型和值约束

大多数约束用于限制路由，因此它们仅匹配具有可转换为指定类型或具有特定格式的片段的URL。本节开头使用的int约束是一个很好的例子：只有当受约束的片段的值可以解析为.NET的int值时，才会匹配路由。代码清单15-28演示了range约束的用法，对路由进行限制，只有当一个片段值可以转换为int类型并且介于指定值之间时才匹配URL。

代码清单15-28　在UrlsAndRoutes文件夹下的Startup.cs文件中基于类型和值进行约束

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
```

```
namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=In
dex}/{id:range(10,20)?}");
            });
        }
    }
}
```

此例中的约束已应用于可选的id片段。如果请求URL少于3个片段，就忽略约束。如果存在id片段，那么仅当片段值可以转换为int类型且介于10～20时，路径才会匹配URL。range约束包含边界值，也就是说，10和20也被认为处在约束范围内。

15.7.3　组合约束

如果需要将多个约束应用于单个片段，可以将它们链接在一起，约束之间用冒号分隔，如代码清单15-29所示。

代码清单15-29　在UrlsAndRoutes文件夹下的Startup.cs文件中组合内联约束

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
```

```
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller=Home}/{actio
n=Index}"
                        + "/{id:alpha:minlength(6)?}");
            });
        }
    }
}
```

在以上代码中，已将alpha和minlength约束应用于id片段。问号表示这是一个在所有约束之后应用的可选片段。这些约束的组合效果是：路由将仅匹配省略了id片段的URL（因为id片段是可选的），或者当id片段存在且包含至少6个字母字符时才匹配。

如果不使用内联约束，则必须使用Microsoft.AspNetCore.Routing.CompositeRouteConstr类，以允许多个约束与匿名类型对象中的单个属性相关联。代码清单15-30显示了在代码清单15-29中使用的组合约束。

代码清单15-30　在UrlsAndRoutes文件夹下的Startup.cs文件中组合单独的约束

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller}/{action}/{i
d?}",
                    defaults: new { controller = "Home"
, action = "Index" },
                    constraints: new {
```

```
                        id = new CompositeRouteConstrai
nt(
                            new IRouteConstraint[] {
                                new AlphaRouteConstrain
t(),

                                new MinLengthRouteConst
raint(6)
                            })
                    });
            });
        }
    }
}
```

CompositeRouteConstraint类的构造函数接收实现了IRouteConstraint对象的枚举，IRouteConstraint对象是定义路由约束的接口。仅当满足所有约束时，路由系统才允许路由匹配URL。

15.7.4　定义自定义约束

如果标准约束无法满足需求，那么可以通过实现Microsoft.AspNetCore.Routing命名空间中定义的IRouteConstraint接口来定义自定义约束。为了演示此功能，在示例项目中添加Infrastructure文件夹，

并创建名为WeekDayConstraint.cs的类文件，内容如代码清单15-31所示。

代码清单15-31　Infrastructure文件夹下的WeekDayConstraint.cs文件的内容

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System.Linq;

namespace UrlsAndRoutes.Infrastructure {
    public class WeekDayConstraint : IRouteConstraint {
        private static string[] Days = new[] { "mon", "
tue", "wed", "thu",
                                                        "fri", "
sat", "sun" };

        public bool Match(HttpContext httpContext, IRou
ter route,
            string routeKey, RouteValueDictionary value
s,
            RouteDirection routeDirection) {

            return Days.Contains(values[routeKey]?.ToSt
ring().ToLowerInvariant());
        }
    }
}
```

IRouteConstraint接口定义了Match方法，可调

用该方法以允许约束来决定请求是否应该与路由匹配。Match方法的参数提供对多种来源的请求的访问，包括客户端、路由、受约束片段的名称、从URL提取的片段变量以及请求是否检查传入或传出的URL（第16章将解释传出的URL）。

在该例中，使用routeKey参数从values参数获取已应用约束的片段变量的值，将它们转换为小写字符串，并查看是否匹配在静态字段Days中定义的一周中的那几天。代码清单15-32使用单独的方法将新约束应用于示例路由。

代码清单15-32　在UrlsAndRoutes文件夹下的Startup.cs文件中应用自定义约束

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller}/{action}/{i
d?}",
                    defaults: new { controller = "Home"
, action = "Index" },
                    constraints: new { id = new WeekDay
Constraint() });
            });
        }
    }
}
```

仅当id片段不存在（例如/Customer/List）或者与约束类中定义的星期几（例

如/Customer/List/Fri）匹配时，路由才会匹配URL。

定义内联自定义约束

为了设置自定义约束并使其可以内联使用，需要执行一个额外的配置步骤，如代码清单15-33所示。

代码清单15-33　在UrlsAndRoutes文件夹下的Startup.cs文件中使用内联自定义约束

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", t
ypeof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=In
dex}/{id:weekday?}");
            });
        }
    }
}
```

以上代码在ConfigureService方法中配置了RouteOptions对象，以控制路由系统的一些行为。ConstraintMap属性会返回一个字典，用于将内联约束的名称转换为提供约束逻辑的IRouteConstraint实现类。向字典添加一个新的映射，将WeekDayConstraint类内联引用为weekday，如下所

示：

```
...
template: "{controller=Home}/{action=Index}/{id:weekday?}",
...
```

约束的效果是相同的，但设置了映射以允许自定义类被内联使用。

15.8 使用特性路由

到目前为止，本章的所有示例都是使用一种称为“基于约定的路由”的技术定义的。MVC还支持称为特性路由的技术，直接使用控制器类的C#属性来定义路由。下面将展示如何使用特性创建和配置路由，这些特性可以与前面示例中的基于约定的路由自由混合使用。

15.8.1 准备特性路由

在Startup.cs文件中调用UseMvc方法时，将启用特性路由。MVC检查应用程序中的控制器类，查找具有特性路由的任何控制器类，并为它们创建路由。

本节中，示例应用程序已还原到默认路由配置，如代码清单15-34所示。

代码清单15-34　使用UrlsAndRoutes文件夹下的Startup.cs文件中的默认路由配置

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
```

```
n services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

默认路由将使用以下模式匹配URL：

```
{controller}/{action}/{id?}
```

15.8.2　应用特性路由

Route 特性用于指定单个控制器和操作方法的路由。在代码清单 15-35 中，已将 Route 特性应用于CustomerController类。

代码清单15-35　在Controllers文件夹下的CustomerController.cs

文件中应用Route特性

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("myroute")]
        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catc
hall"];
            return View("Result", r);
        }
    }
}
```

Route特性可用于为操作方法或控制器定义路由。在以上代码中，已将Route特性应用于Index操

作方法，并将myroute指定为应该使用的路由。效果是更改了用于访问Customer控制器定义的操作方法的路由集合，如表15-9所示。

表**15-9**　路由集合

路由	描述
/Customer/List	以List操作方法为目标，依赖于Startup.cs中的默认路由
/myroute	以Index操作方法为目标

有两点需要注意。第一点是当使用Route特性时，你提供的用于配置特性的值将用于定义完整的路由，因此myroute成为访问Index操作方法的完整URL。需要注意的第二点是，使用Route特性可以防止使用默认路由配置，因此无法再使用/Customer/Index URL访问Index操作方法。

1．更改操作方法的名称

为单个操作方法定义唯一路由在大多数应用程序中没什么用，但Route特性也可以更灵活地使用。代码清单15-36在路由中使用特殊的[controller]标记来引用控制器并设置路由的基本部分。

提　示

使用ActionName属性也可以更改操作的名称，具体将在第31章中介绍。

代码清单15-36　重命名Controllers文件夹下的CustomerController.cs文件中的操作方法

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;
```

```
namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("[controller]/MyAction")]
        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catc
hall"];
            return View("Result", r);
        }
    }
}
```

在Route特性的参数中使用[controller]标记就像使用nameof表达式一样，允许指定到控制器的路由，而无须对类名进行硬编码。表15-10描述了代码清单15-36中路由的效果。

表**15-10　Customer**控制器的路由

路由	描述
/Customer/List	以List操作方法为目标
/Customer/MyAction	以Index操作方法为目标

2．创建更复杂的路由

Route特性也可以应用于控制器类，允许定义路由的结构，如代码清单15-37所示。

代码清单15-37　在Controllers文件夹下的CustomerController.cs文件中应用Route特性

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    [Route("app/[controller]/actions/[action]/{id?}")]
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
```

```
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catc
hall"];
            return View("Result", r);
        }
    }
}
```

以上路由混合使用了静态片段和变量片段，并使用[controller]和[action]标记来分别引用控制器类的名称和操作方法。表15-11显示了代码清单15-37中路由的效果。

表**15-11**　代码清单**15-37**中路由的效果

路由	描述
app/customer/actions/index	以Index操作方法为目标

app/customer/actions/index/myid	以Index操作方法为目标，并将可选的id片段设置为myid
app/customer/actions/list	以List操作方法为目标
app/customer/actions/list/myid	以List操作方法为目标，并将可选的id片段设置为myid

15.8.3　应用路由约束

使用特性定义的路由可以像Startup.cs文件中定义的路由那样，使用与基于约定的路由相同的内联技术进行约束。在代码清单15-38中，已将本章前面创建的自定义约束应用于Route特性定义的可选id片段。

代码清单15-38　在Controllers文件夹下的CustomerController.cs文件中约束路由

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    [Route("app/[controller]/actions/[action]/{id:weekd
ay?}")]
```

```
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController)
,
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catc
hall"];
            return View("Result", r);
        }
    }
}
```

可以使用表15-8中描述的所有约束，也可如代码所示，使用已在RouteOptions服务中注册的自定义约束。可以通过将它们链接在一起并用冒号分隔它们来应用多个约束。

15.9 小结

本章深入介绍了路由系统。你已经了解了按约定或特性定义路由的方式。你还了解了如何匹配和处理传入的URL，以及如何通过更改匹配URL片段的方式和使用默认值及可选片段来自定义路由。这里还展示了如何使用内置约束和自定义约束，用来约束路由以缩小它们匹配的请求范围。

下一章将展示如何通过路由在视图中生成传出的URL，以及如何使用area特性，它依赖路由系统，可以用于管理大型的和复杂的MVC应用程序。

第16章　高级路由特性

上一章展示了如何使用路由系统来处理传入的URL，我们还需要能够使用URL模式来生成可以签入视图中的输出URL，以便用户可以单击链接，以正确的控制器和action为目标，将表单回递给应用程序。

本章将展示生成输出URL的不同技术，展示如何通过替换标准的MVC路由实现类来定制路由系统以及使用MVC区域（area）特性，area特性让你能够将大型的复杂MVC应用程序分解为若干可管理的小型区域。最后，以一些有关MVC应用程序中URL模式最佳实践的建议结束本章，表16-1汇总了在上下文中使用高级路由特性时的一些问题。

表**16-1**　在上下文中使用高级路由特性

问题	答案
路由是什么?	路由系统除了为HTTP请求匹配URL外，还提供了很多功能，比如支持在视图中生成URL，使用自定义类替换内置的路由功能，在应用程序中构建相互隔离的模块，等等
为什么路由很有用?	路由的每个功能在它们各自的应用场景中都是很有用的，比如通过设置路由格式，我们不用更新视图，就可以很容易地更改URL的格式，使用自定义类可以让我们根据需要定制路由系统，更容易地利用应用程序构建复杂的系统
如何使用路由？	有关路由使用的详细信息，请参阅本章
路由有什么缺点或限制吗？	复杂应用程序的路由配置可能变得难以管理
还有其他的选择么？	没有，路由系统是ASP.NET的组成部分之一

表16-2列出了本章要介绍的操作。

表**16-2**　本章要介绍的操作

操作	方法	代码清单

使用URL生成锚点元素	使用asp-action和asp-controller属性	代码清单16-2～代码清单16-5
提供路由片段的值	使用以asp-route为前缀的属性	代码清单16-6和代码清单16-7
生成完全限定的URL	使用asp-procotol、asp-host和asp-fragment属性	代码清单16-8
选择生成URL的路由	使用asp-route属性	代码清单16-9和代码清单16-10
生成没有HTML元素的URL	在视图或操作方法中使用Url.Action方法	代码清单16-11和代码清单16-12
自定义路由系统	在Startup类中使用Configure方法	代码清单16-13
创建自定义路由类	实现IRouter接口	代码清单16-14～代码清单16-21
将应用程序分解为各个功能模块	利用Area属性创建区域	代码清单16-22～代码清单16-28

16.1　准备示例项目

本章继续使用前一章的UrlsAndRoutes项目，但

是在Startup类中进行一些修改。在Startup类中，使用具有相同效果的显式路由替换UseMvcWithDefaultRoute方法，如代码清单16-1所示。

代码清单16-1　修改UrlsAndRoutes文件夹下的Startup类中的路由配置

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
            services.AddMvc();
```

```
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

当启动程序时，浏览器会请求默认的URL，请求会被发送给Home控制的Index操作方法，如图16-1所示。

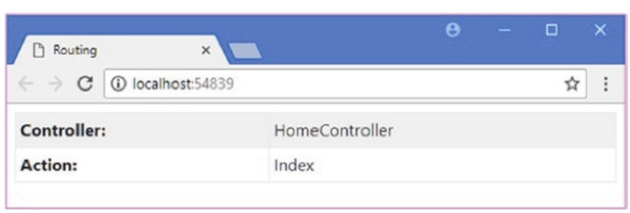


图16-1　运行示例应用程序

16.2　在视图中生成传出的URL

在几乎每个MVC应用程序中，都希望用户可以从一个视图跳转到另一个视图，这通常依赖于在第一个视图中包含一个针对生成第二个视图的操作方法的链接。方法是在页面中添加一个静态元素（称为锚点元素），用href属性定位到操作方法，如下所示：

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

如果应用程序使用默认的路由配置，这个HTML元素将创建一个链接，指向Home控制器中名为CustomVariable的操作方法。但是这样手动生成URL是非常危险的，当更改应用程序的URL格式时，必须查看所有视图来更新所有指向控制器和操作方法的链接，这将是一个非常冗长乏味且容易出错的过程，还难以测试。所以，更好的选择是使用路由系统来生成传出的URL，这样可以确保使用程

序的URL格式来动态生成URL，并且还能在一定程度上反映程序的URL格式。

16.2.1　创建传出的链接

在视图中生成传出的URL的最简单方法是使用锚标签助手，它将为HTML元素生成href属性，如代码清单16-2所示，可在视图/Views/Shared/Result.cshtml中使用锚标签助手来创建传出的链接。

提　示

第23章将解释标签助手的工作原理。

代码清单16-2　在Views/Shared文件夹下的Result.cshtml文件中使用锚标签助手

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <a asp-action="CustomVariable">This is an outgoing
URL</a>
</body>
</html>
```

asp-action属性用来指定href属性中的URL指向的对应名称的操作方法。可以通过启动应用程序来查看结果，如图16-2所示。

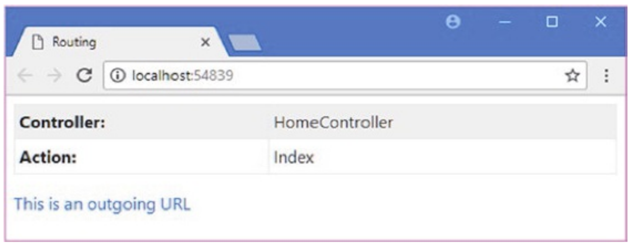


图16-2　使用标签助手来生成链接

标签助手使用当前路由配置在a元素上设置href属性。如果查看浏览器中的HTML标记，你会看到其中包含以下元素：

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

这个链接看起来和之前手动编写的链接是一样的，而且为了生成这个链接，我们还做了很多其他的工作。但是这种方法的好处是链接会自动响应路由配置的更改。为了演示，在Startup.cs文件中添加

一个新的路由配置，如代码清单16-3所示。

代码清单16-3　在UrlsAndRoutes文件夹下的Startup.cs文件中添加路由配置

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
```

```
            app.UseStaticFiles();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: "NewRoute",
                    template: "App/Do{action}",
                    defaults: new { controller = "Home"
 });

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

新路由更改了针对Home控制器的请求的URL模式。如果启动应用，你将看到这些更改会反映在由HTML辅助方法ActionLink生成的HTML标记中，如下所示：

```
<a href="/App/DoCustomVariable">This is an outgoing URL
</a>
```

使用标签助手生成链接可解决重要的维护问题。当更改路由格式时，视图中的传出链接会自动

根据路由格式更改，无须在视图中手动编辑。

当单击链接时，传出的URL用于创建传入的HTTP请求，然后使用相同的路由来定位将用于处理请求的控制器和操作方法，如图16-3所示。

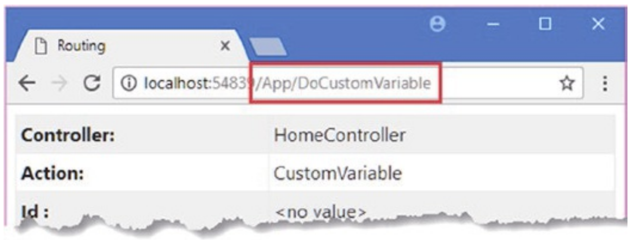


图16-3　单击链接，将传出的URL转换为传入的请求

理解传出的URL的路由匹配

你已经看到了通过更改路由配置的URL模式，可以修改传出的URL的生成方式。应用程序通常会定义多个路由，所以了解如何为URL的生成选择路由规则很重要。路由系统按照定义的顺序处理路

由，依次检查每个路由，查看是否匹配，这需要满足以下3个条件。

- URL模式中定义的每个片段变量都必须有一个值。为了查找每个片段变量的值，路由系统首先查看你提供的值（使用匿名类型的属性），然后查看当前请求的变量值，最后查看路由中定义的默认值（本章稍后再回到这些值的第二个来源）。
- 为片段变量提供的任何值都不会与路由中定义的默认变量发生不一致。这些是已经提供了默认值但在URL模式中不会出现的变量。例如，在下面的路由定义中，myVar是默认变量。

```
     routes.MapRoute("MyRoute", "{controller}/{action}"
,
         new { myVar = "true" });
```

为了使以上路由匹配，必须注意不要为myVar提供值，或者确保提供的值与默认值匹配。

- 所有片段变量的值必须满足路由约束。有关不同种类约束的示例，请参阅15.6节。

需要明确的是，路由系统不会尝试选取最佳的路由规则来进行匹配，而只是寻找第一个可以匹配的规则，并根据该规则生成URL，任何后续的路由规则都会被忽略。因此，必须首先定义那些特殊的路由规则，检查生成的传出URL也很重要。如果尝试生成找不到匹配路由的URL，将创建如下包含空的href属性的链接：

```
<a href="">This is an outgoing URL</a>
```

以上链接将正确显示在视图中，但在用户单击时不会按预期方式运行。如果只生成URL（本章稍后会展示如何实现），那么结果将为null，在视图中作为空字符串呈现。可以使用命名路由对路由匹配进行一些控制。

1．定位其他控制器

当在一个元素上指定asp-action属性时，标签助手会在渲染视图时假定你要定位到同一控制器中的action。要创建以不同控制器为目标的传出URL，可以使用asp-controller属性，如代码清单16-4所示。

代码清单16-4　在Views/Shared文件夹下的Result.cshtml文件中定位不同的控制器

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
```

```
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <a asp-controller="Admin" asp-action="Index">
        This targets another controller
    </a>
</body>
</html>
```

在渲染视图时，你将看到生成了以下HTML标记：

```
<a href="/Admin">This targets another controller</a>
```

指向Admin控制器中Index操作方法的请求URL被标签助手表示为/Admin。路由系统知道应用程序中定义的路由默认使用Index操作方法，因而允许忽略不必要的片段变量。

路由系统包含那些使用 Route 特性给操作方法指定路由规则的路由。在代码清单 16-5 中，asp-controller属性用于定位在第15章中使用了Route特性的Customer控制器中的Index操作方法。

代码清单16-5　在Views/Shared文件夹下的Result.cshtml文件中定位操作方法

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="panel-body">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
```

```
        }
    </table>
    <a asp-controller="Customer" asp-action="Index">This is an outgoing URL</a>
</body>
</html>
```

生成的链接如下：

```
<a href="/app/Customer/actions/Index">This is an outgoing URL</a>
```

这对应于第15章中为Customer控制器设置的Route特性：

```
...
[Route("app/[controller]/actions/[action]/{id:weekday?}")]
public class CustomerController : Controller {
...
```

2．传递额外的变量值

可以使用asp-route-片段变量名格式将片段变量的值传递给路由系统，比如可以使用asp-route-id将

值传给路由系统中的id，如代码清单16-6所示。

代码清单16-6　为Views/Shared文件夹下的Result.cshtml文件中的片段变量提供值

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <a asp-controller="Home" asp-action="Index" asp-rou
te-id="Hello">
        This is an outgoing URL
    </a>
```

```
</body>
</html>
```

这里为名为id的片段变量设置了值，如果应用代码清单16-6所示的路由，在视图渲染的时候将会呈现以下HTML标记：

```
<a href="/App/DoIndex?id=Hello">This is an outgoing URL
</a>
```

请注意，片段变量的值已经根据路由规则做了设置，成为URL中查询字符串的一部分。路由中没有设置对应于id的片段变量，为了解决这个问题，修改Startup.cs文件中的路由，只留下一个拥有id片段变量的路由，如代码清单16-7所示。

代码清单16-7　在UrlsAndRoutes文件夹下的Startup.cs文件中修改路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

                //routes.MapRoute(
                //    name: "NewRoute",
                //    template: "App/Do{action}",
                //    defaults: new { controller = "Hom
e" });

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
```

```
            });
        }
    }
}
```

再次运行应用，你将看到标签助手生成的URL中已经包含了id属性的值。

```
<a href="/Home/Index/Hello">This is an outgoing URL</a>
```

理解片段变量的重用

之前在描述匹配的路由传出URL的方式时，本书解释说，当试图为路由的URL模式中的每个片段变量查找值时，路由系统将查看当前请求的值。这种方式可能会困扰很多程序员，并且导致冗长的调试会话。

假设应用程序有如下路由：

```
...
app.UseMvc(routes => {
    routes.MapRoute(name: "MyRoute",
        template: "{controller}/{action}/{color}/{page}
");
});
...
```

假设我们所在的页面URL为/Home/Index/Red/100，在页面中利用以下代码渲染链接：

```
...
<a asp-controller="Home" asp-action="Index" asp-route-p
age="789">
    This is an outgoing URL
</a>
...
```

你可能期望路由系统无法匹配路由，因为没有为color片段变量提供值，也没有定义默认值。期望是错误的，路由系统将与定义的路由匹配，还将生成以下HTML标记：

```
...
<a href="/Home/Index/Red/789">This is an outgoing URL</
```

```
a>
...
```

路由系统热衷于对路由进行匹配，在生成传出的URL时，将在传入的URL中重用片段变量的值。在这种情况下，根据页面使用的URL，路由系统将设置color变量的值为Red。

这并不是最后的手段。路由系统将应用此技术作为对路由的常规评估的一部分，即使后面的路由不需要重用当前请求中的值，也会进行匹配。

强烈建议你不要依赖此行为，并为URL模式中的所有片段变量提供值。依赖这种行为不仅会使代码更难阅读，而且还会对用户发出请求的顺序做出假设，这将在应用程序进入维护时令人抓狂。

3．生成完全限定的URL

到目前为止，生成的所有链接都包含相对URL，但是锚标签助手也可以生成完全限定的URL，如代码清单16-8所示。

代码清单16-8　在Views/Shared文件夹下的Result.cshtml文件中生成完全限定的URL

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
```

```
    <a asp-controller="Home" asp-action="Index" asp-route-id="Hello"
       asp-protocol="https" asp-host="myserver.mydomain.com"
       asp-fragment="myFragment">
        This is an outgoing URL
    </a>
</body>
</html>
```

asp-protocol、asp-host与asp-fragment属性分别用于指定协议（https）、服务器名称（myserver.mydomain.com）和URL片段（myFragment）。将这些值与路由系统的输出相结合，即可创建完全限定的URL，可以运行应用以查看发送到浏览器的HTML：

```
<a href="https://myserver.mydomain.com/Home/Index/Hello#myFragment">
    This is an outgoing URL
</a>
```

使用完全限定的URL时要小心，因为它们创建了与应用程序基础架构的依赖关系，并且基础架构

发生变化时，必须记住对MVC视图进行相应的更改。

4．从特定路由生成URL

在前面的示例中，我们演示了路由系统如何选择路由来生成URL。当需要生成特定格式的URL时，可以指定用于生成URL的路由。为了演示这是如何工作的，这里向Startup.cs文件添加一个新的路由，这时在示例应用程序中有两个路由，如代码清单16-9所示。

代码清单16-9　在UrlsAndRoutes文件夹的Startup.cs文件中添加路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

                //routes.MapRoute(
                //    name: "NewRoute",
                //    template: "App/Do{action}",
                //    defaults: new { controller = "Hom
e" });

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");

                routes.MapRoute(
                    name: "out",
                    template: "outbound/{controller=Hom
```

```
e}/{action=Index}");
            });
        }
    }
}
```

代码清单16-10所示的视图中包含两个锚元素，每个都指定了相同的控制器和操作。不同之处在于第二个元素使用asp-route属性来指定使用的路由。

代码清单16-10　在Views/Shared文件夹下的Result.Cshtml文件中生成URL

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
```

```
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <a asp-controller="Home" asp-action="CustomVariable
">This is an outgoing URL</a>
    <a asp-route="out">This is an outgoing URL</a>
</body>
</html>
```

只有在asp-controller属性和asp-action属性不存在时，才可以使用asp-route属性，这意味着只能为渲染视图的控制器和操作选择特定的路由。如果运行示例程序并请求URL /Home/CustomVariable，你将看到路由会生成两个不同的URL：

```
<a href="/Home/CustomVariable">This is an outgoing URL<
/a>
<a href="/outbound">This is an outgoing URL</a>
```

针对命名路由的情况

依靠路由名称生成传出URL的问题在于，这样做会破坏对MVC设计模式至关重要的关注点分离原则。在视图或操作方法中生成链接或URL时，需要关注用户将被引导到的操作和控制器，而不是要使用的URL格式。通过将不同路由引入视图或控制器，可以避免创建依赖关系。在作者自己的项目中，作者倾向于避免命名路由（通过为name参数指定null），并且更喜欢使用代码注释来提醒自己每个路由的目的。

16.2.2　创建非链接的URL

标签助手的限制是：对于转换的HTML元素，当需要为应用程序生成URL而不想使用周围的HTML时，无法轻易地重用。

MVC提供了一个辅助类，可以直接使用Url.Action方法创建URL，如代码清单16-11所示。

代码清单16-11　在Views/Shared文件夹下的Result.cshtml文件中生成URL

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <p>URL: @Url.Action("CustomVariable", "Home", new {
 id = 100 })</p>
```

```
</body>
</html>
```

以上代码在Url.Action方法的参数中指定了操作方法、控制器和片段变量的值，生成的结果如下：

```
<p>URL: /Home/CustomVariable/100</p>
```

在操作方法中生成URL

Url.Action方法也可以用于在操作方法中使用C#代码来创建URL。代码清单16-12修改了Home控制器的一个操作方法，并使用Url.Action生成了一个URL。

代码清单16-12　在Controllers文件夹下的HomeController.cs文件里的操作方法中生成URL

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
```

```
    public class HomeController : Controller {
        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id ?? "<no value>";
            r.Data["Url"] = Url.Action("CustomVariable"
, "Home", new { id = 100 });
            return View("Result", r);
        }
    }
}
```

如果运行示例程序并且请求URL /Home/CustomVariable/100，你将看到表格中有一行显示了这个URL，如图16-4所示。

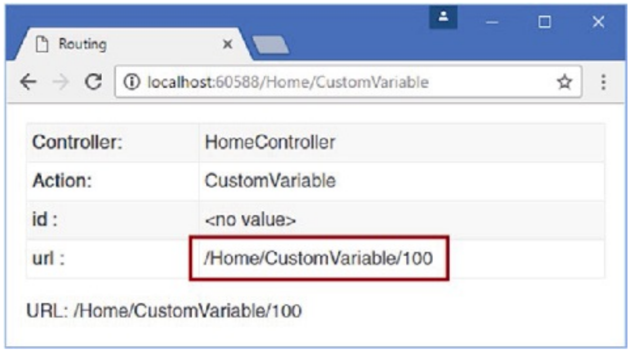


图16-4　在操作方法中生成URL

16.3　自定义路由系统

你已经看到了路由系统的灵活性和可配置性，如果这些还不符合要求，你也可以自定义行为。本节将展示集中不同的自定义路由的方法。

16.3.1　更改路由系统配置

第15章展示了如何在Startup.cs文件中配置RouteOptions对象以设置自定义路由约束。使用RouteOptions对象的属性（见表16-3），配置一些路由功能。

表**16-3　RouteOptions**对象的属性

属性	描述
AppendTrailingSlash	当为true时，为路由系统生成的URL附加尾部斜线，默认值为false
LowercaseUrls	当为true时，如果控制器、action或片段变量的值包含大写字母，就将URL转换为小写，默认值为false

代码清单16-13向Startup.cs文件添加了路由配置，以设置表16-3中描述的两个配置属性。

代码清单16-13　在UrlsAndRoutes文件夹下的Startup.cs文件中配置路由系统

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
```

```
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
 {
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint));
                options.LowercaseUrls = true;
                options.AppendTrailingSlash = true;
            });
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");

                routes.MapRoute(
                    name: "out",
                    template: "outbound/{controller=Hom
e}/{action=Index}");
            });
        }
    }
}
```

如果运行应用并检查路由系统生成的URL，你将看到URL全部为小写，并且附加了尾部斜杠，如图16-5所示。

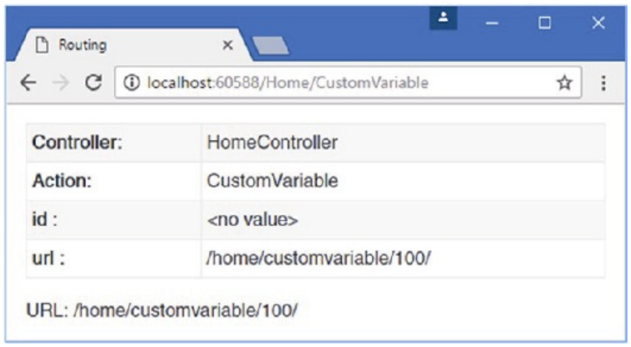


图16-5　配置路由系统

16.3.2　创建自定义路由类

如果不喜欢路由系统匹配URL的方式，或者需要在应用中进行一些特殊的实现，可以创建自己的路由类并使用它们来处理URL。ASP.NET提供了Microsoft.AspNetCore.Routing.IRouter接口，可以通过实现此接口来创建自定义路由。以下是IRouter接口的定义：

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Routing {

    public interface IRouter {

        Task RouteAsync(RouteContext context);

        VirtualPathData GetVirtualPath(VirtualPathConte
xt context);
    }
}
```

要创建自定义路由，可以实现RouteAsync方法来处理传入的请求，并通过实现GetVirtualPath方法来生成传出的URL。

为了演示，我们将创建一个可以处理传统URL请求的自定义路由类。想象一下，这里已经将一个现有的应用程序迁移到MVC，但是有些用户已经将之前的网址加入书签，或者硬编码到脚本中。为了仍然支持那些旧的URL，可以使用常规路由系统，但是这个问题的解决方法值得参考。

1．路由传入的URL

要了解自定义路由如何工作，可以首先创建一个路由来处理请求，而不使用控制器和视图。在Infrastructure文件夹中创建一个名为LegacyRoute.cs的类文件，用于实现IRouter接口，如代码清单16-14所示。

代码清单16-14　Infrastructure文件夹下的LegacyRoute.cs文件的内容

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UrlsAndRoutes.Infrastructure {
    public class LegacyRoute : IRouter {
        private string[] urls;

        public LegacyRoute(params string[] targetUrls)
{
            this.urls = targetUrls;
        }
```

```
        public Task RouteAsync(RouteContext context) {

            string requestedUrl = context.HttpContext.R
equest.Path
                .Value.TrimEnd('/');

            if (urls.Contains(requestedUrl, StringCompa
rer.OrdinalIgnoreCase)) {
                context.Handler = async ctx => {
                    HttpResponse response = ctx.Respons
e;
                    byte[] bytes = Encoding.ASCII.GetBy
tes($"URL: {requestedUrl}");
                    await response.Body.WriteAsync(byte
s, 0, bytes.Length);
                };
            }
            return Task.CompletedTask;
        }

        public VirtualPathData GetVirtualPath(VirtualPa
thContext context) {
            return null;
        }
    }
}
```

LecagyRoute类实现了IRouter接口，但只定义了用于处理传入请求的RouteAsync方法；我们将在后面的内容中添加处理传出URL的方法。

RouteAsync方法中只有几条语句，但它们的工作依赖于一些重要的ASP.NET类型。我们先从方法签名开始介绍：

```
...
public async Task RouteAsync(RouteContext context) {
...
```

RouteAsync方法负责评估是否可以处理请求，如果可以，就可以通过生成响应并发送回客户端来管理整个过程。因为RouteAsync方法返回一个Task对象，所以这个过程是异步执行的。

RouteAsync方法使用了RouteContext参数，从而提供对请求的所有已知信息的访问，并提供将响应发送回客户端所需的功能。RouteContext类在命名空间Microsoft.AspNetCore.Routing中定义，有3个属性，如表16-4所示。

表**16-4　RouteContext**类中定义的属性

属性	描述
RouteData	返回一个Microsoft.AspNetCore.Routing.RouteData对象，当编写依赖于MVC功能的自定义路由时，该对象用于定义控制器、操作方法以及用于处理请求的参数
HttpContext	返回一个Microsoft.AspNetCore.Http.HttpContext对象，该对象提供了访问HTTP请求的详细信息以及生成HTTP响应的方法
Handler	用于为路由系统提供处理请求的RequestDelegate。如果未设置该属性，路由系统将继续按照应用设置的路由集合工作

路由系统调用RouteAsync方法来处理应用中的每个路由，并在每次调用后检查Handler属性的值。如果Handler属性已设置为RequestDelegate，路由就为路由系统提供可以处理请求的委托，并调用委托以生成响应。以下是RequestDelegate的签名，RequestDelegate定义在Microsoft.AspNetCore.Http命名空间中。

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Http {
    public delegate Task RequestDelegate(HttpContext co
```

```
ntext);
}
```

RequestDelegate接收HttpContext对象并返回一个任务来生成响应。如果没有为任何路由设置Handler属性，路由系统便得知应用程序无法处理请求，因而生成404-Not Found响应。

考虑到这一点，RouteAsync方法的实现必须确定是否可以处理通常需要HttpContext的请求。在该例中，使用了HttpContext.Request属性，该属性返回描述请求的Microsoft.AspNetCore.Http.HttpRequest对象。HttpRequest对象提供对有关请求的所有可用信息的访问，包括头文件、主体以及请求发起地址的详细信息。Path属性最有趣，因为它提供了客户端发送的URL请求的详细信息。Path属性会返回一个PathString对象，该对象提供了一些有用的方法来编写和比较URL路径。但这里使用了Value属性，因

为它将URL的整个路径部分作为字符串提供了，从而可以与支持的URL集合进行比较。URL可由LegacyRoute构造函数接收。

```
...
string requestedUrl = context.HttpContext.Request.Path.
Value.TrimEnd('/');
if (urls.Contains(requestedUrl, StringComparer.OrdinalI
gnoreCase)) {
...
```

可以由用户手动添加，也可以使用AppendTrailingSlash配置选项进行添加。因为在URL路径的最后会有斜杠，所以需要使用TrimEnd方法来去除URL尾部的斜杠。

如果请求的路径已由LegacyRoute配置为支持路径，那么将使用一个用于生成响应的Lambda函数来设置Handler属性，如下所示：

```
...
context.Handler = async ctx => {
    HttpResponse response = ctx.Response;
```

```
    byte[] bytes = Encoding.ASCII.GetBytes($"URL: {requestedUrl}");
    await response.Body.WriteAsync(bytes, 0, bytes.Length);
};
...
```

HttpContext.Response属性会返回一个HttpResponse对象，该对象可用于创建对客户端的响应，以及提供对发送到客户端的头文件和内容的访问。这里使用HttpResponse.Body.WriteAsync方法异步写入一个简单的ASCII字符串作为响应，虽然在正式的项目中不应该这么做，但这允许生成响应，而不必选择和渲染视图（下一章会介绍如何在MVC中选择可渲染视图）。

当Handler属性被设置时，路由系统知道对路由的搜索已经完成，并且可以调用委托来生成对客户端的响应。

1）应用自定义路由类

到目前为止，一直用于创建路由的MapRoute扩展方法不支持使用自定义路由类。要使用LegacyRoute类，必须采取不同的方法，如代码清单16-15所示。

代码清单16-15　在UrlsAndRoutes文件夹下的Startup.cs文件中应用自定义路由类

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
 {
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint));
```

```
                options.LowercaseUrls = true;
                options.AppendTrailingSlash = true;
            });
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.Routes.Add(new LegacyRoute(
                    "/articles/Windows_3.1_Overview.htm
l",
                    "/old/.NET_1.0_Class_Library"));

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");

                routes.MapRoute(
                    name: "out",
                    template: "outbound/{controller=Hom
e}/{action=Index}");
            });
        }
    }
}
```

使用自定义路由类时，必须使用路由集合上的Add方法来注册IRouter实现类。在本例中，

LegacyRoute构造函数的参数是希望自定义路由支持的旧URL。可以通过启动应用程序并请求/articles/Windows_3.1_Overview.html来查看效果。自定义路由会显示请求的URL，如图16-6所示。



图16-6　使用自定义路由

2）路由到MVC控制器

在简单的URL匹配字符串和使用MVC系统的控制器、操作和Razor视图之间还有很大的差距。但幸运的是，不必在创建自定义路由时实现这些功能，MVC会在后台完成这些重要工作。为了使用MVC的基础架构，在Controllers文件夹中添加一个名为LegacyController.cs的类文件并用它定义控制

器，如代码清单16-16所示。

代码清单16-16　Controllers文件夹下的LegacyController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace UrlsAndRoutes.Controllers {

    public class LegacyController : Controller {

        public ViewResult GetLegacyUrl(string legacyUrl
)
             => View((object)legacyUrl);
    }
}
```

在这个控制器中，操作方法GetLegacyUrl用来接收客户端发来的包含旧URL的参数。在真实的项目中，在这个控制器的操作方法中应该取回请求的文件，但这里只是在视图中显示URL。

提　示

在代码清单16-16中，为View方法传值的参数为object类型。View方法的一个重载版本可以使用一个字符串来指定要呈现的视图的名称，并且没有发生转换，C#编译器认为这正是我们想要的重载版本。为了避免这种情况，需要转换为object类型，以便显式地调用传递视图模型的重载版本，并使用默认视图。也可以通过使用同时具有视图名称和视图模型的重载版本来解决这个问题，但是这里不希望在操作方法和视图之间明确地进行关联。详细信息请参见第17章。

创建Views/Legacy文件夹，并添加一个名为

GetLegacyUrl.cshtml的视图，如代码清单16-17所示。视图将显示模型的值，在这里显示的是客户端请求的URL。

代码清单16-17　Views / Legacy文件夹下的GetLegacyUrl.cshtml文件的内容

```
@model string
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <h2>GetLegacyURL</h2>
    The URL requested was: @Model
</body>
</html>
```

在代码清单16-18中，更新LegacyRoute类，从而将处理的URL路由到Legacy控制器中的

GetLegacyUrl操作方法。

代码清单16-18　在LegacyRoute.cs文件中将URL路由到控制器

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Internal;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes.Infrastructure {
    public class LegacyRoute : IRouter {
        private string[] urls;
        private IRouter mvcRoute;

        public LegacyRoute(IServiceProvider services, p
arams string[] targetUrls) {
            this.urls = targetUrls;
            mvcRoute = services.GetRequiredService<MvcR
outeHandler>();
        }

        public async Task RouteAsync(RouteContext conte
xt) {

            string requestedUrl = context.HttpContext.R
equest.Path
                .Value.TrimEnd('/');

            if (urls.Contains(requestedUrl, StringCompa
```

```
rer.OrdinalIgnoreCase)) {
                context.RouteData.Values["controller"]
= "Legacy";
                context.RouteData.Values["action"] = "G
etLegacyUrl";
                context.RouteData.Values["legacyUrl"] =
 requestedUrl;
                await mvcRoute.RouteAsync(context);
            }
        }

        public VirtualPathData GetVirtualPath(VirtualPa
thContext context) {
            return null;
        }
    }
}
```

Microsoft.AspNetCore.Mvc.Internal.MvcRouteHa类提供了使用controller和action片段变量定位控制器类、执行操作方法并将结果返回客户端的机制。这个类可以由自定义的IRouter实现来调用，IRouter实现提供了controller和aciton的值以及所需的任何其他值，比如操作方法的参数。

代码清单16-18创建了一个MvcRouteHandler实

例，以把任务委派给控制器。为了实现这些，需要给路由提供数据，如下所示：

```
...
context.RouteData.Values["controller"] = "Legacy";
context.RouteData.Values["action"] = "GetLegacyUrl";
context.RouteData.Values["legacyUrl"] = requestedUrl;
...
```

RouteContext.RouteData.Values属性会返回一个用于向MvcRouteHandler类提供数据值的字典。在默认路由系统中，可通过将URL模式应用到请求来创建数据值。但在自定义路由类中，已经对值进行了硬编码，以便始终将目标定位到旧版控制器上的GetLegacyUrl操作。不同请求之间唯一更改的是legacyUrl数据值，它被设置为请求URL，并将被用作由操作方法接收的相同名称的参数。

在代码清单16-18的最后，查找和使用控制器类来处理请求：

```
...
await mvcRoute.RouteAsync(context);
...
```

包含controller、action和legacyUrl值的RouteContext对象将被传递给MvcRouteHandler对象的RouteAsync方法，该方法负责对请求做进一步处理，包括设置Handler属性。因此，LegacyRoute类可以专注于决定处理哪些URL，而不会陷入直接使用控制器的细节。

在本例中，MvcRouteHandler对象必须作为服务进行请求，第18章会解释原因。为了向LegacyRoute构造函数提供用于创建MvcRouteHandler的IServiceProvider对象，可以更新路由的定义语句，以便在Startup类中提供对应用程序服务的访问，如代码清单16-19所示。

代码清单16-19　在UrlsAndRoutes文件夹下的Startup类中提供对应用程序服务的访问

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
 {
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint));
                options.LowercaseUrls = true;
                options.AppendTrailingSlash = true;
            });
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.Routes.Add(new LegacyRoute(
                    app.ApplicationServices,
```

```
                    "/articles/Windows_3.1_Overview.htm
l",
                    "/old/.NET_1.0_Class_Library"));

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");

                routes.MapRoute(
                    name: "out",
                    template: "outbound/{controller=Hom
e}/{action=Index}");
            });
        }
    }
}
```

如果再次启动应用程序并请求地址/articles/Windows_3.1_Overview.html，你将看到简单的文本响应现已被视图的输出替换，如图16-7所示。



图16-7　委托处理控制器和操作

2．创建传出的URL

为了支持生成传出的URL，需要在LegacyRoute类中实现GetVirtualPath方法，如代码清单16-20所示。

代码清单16-20　在Infrastructure文件夹下的LegacyRoute.cs文件中生成传出的URL

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Internal;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes.Infrastructure {
    public class LegacyRoute : IRouter {
        private string[] urls;
        private IRouter mvcRoute;

        public LegacyRoute(IServiceProvider services, p
arams string[] targetUrls) {
            this.urls = targetUrls;
            mvcRoute = services.GetRequiredService<MvcR
outeHandler>();
        }
```

```
        public async Task RouteAsync(RouteContext conte
xt) {

            string requestedUrl = context.HttpContext.R
equest.Path
                .Value.TrimEnd('/');

            if (urls.Contains(requestedUrl, StringCompa
rer.OrdinalIgnoreCase)) {
                context.RouteData.Values["controller"]
= "Legacy";
                context.RouteData.Values["action"] = "G
etLegacyUrl";
                context.RouteData.Values["legacyUrl"] =
 requestedUrl;
                await mvcRoute.RouteAsync(context);
            }
        }

        public VirtualPathData GetVirtualPath(VirtualPa
thContext context) {
            if (context.Values.ContainsKey("legacyUrl")
) {
                string url = context.Values["legacyUrl"
] as string;
                if (urls.Contains(url)) {
                    return new VirtualPathData(this, ur
l);
                }
            }
            return null;
        }
    }
}
```

路由系统调用Startup类中定义的每个路由的GetVirtualPath方法，使每个路由都有机会生成应用程序需要的传出URL。GetVirtualPath方法的参数是一个VirtualPathContext对象，用于提供有关所需URL的信息。表16-5介绍了VirtualPathContext类的属性。

表**16-5　VirtualPathContext**类的属性

属性	描述
RouteName	返回路由的名称
Values	返回可用于片段变量的所有值的字典，并按名称进行索引
AmbientValues	返回有助于生成URL的值的字典，但不会将它们合并到结果中。当实现自己的路由类时，这个字典通常为空
HttpContext	返回一个HttpContext对象，用于提供有关请求及响应的信息

在本例中，使用Values属性获取一个名为legacyUrl的值，如果能匹配路由配置支持的URL，

就返回一个VirtualPathData对象，从而为路由系统提供URL的详细信息。VirtualPathData构造函数的参数是生成URL和URL本身的IRouter。

```
...
return new VirtualPathData(this, url);
...
```

代码清单16-21修改了Result.cshtml视图文件，以要求以自定义视图为目标的传出URL。

代码清单16-21　在Views/Shared文件夹下的Result.cshtml文件中使用自定义路由类生成传出的URL

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
```

```
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td
></tr>
        }
    </table>
    <a asp-route-legacyurl="/articles/Windows_3.1_Overv
iew.html"
       class="btn btn-primary">
        This is an outgoing URL
    </a>
    <p>
        URL: @Url.Action(null, null,
           new { legacyurl = "/articles/Windows_3.1_Ove
rview.html" })
    </p>
</body>
</html>
```

在本例中，不需要为标签助手的传出路由指定控制器和操作，因为它们不在URL生成中使用。考虑到这一点，在a元素中省略了asp-controller和asp-action标签助手属性。当生成URL时，出于相同的原因，将Url.Action辅助方法的前两个参数设置为null。

如果运行应用程序并在默认URL的响应中查看HTML，你将看到自定义路由类已用于创建URL，如下所示：

```
<a class="btn btn-primary" href="/articles/windows_3.1_
overview.html/">
    This is an outgoing URL
</a>
<p>URL: /articles/windows_3.1_overview.html/</p>
```

附加到URL尾部的斜杠是将Startend.cs文件中的AppendTrailingSlash配置选项设置为true的结果，并且需要注意的是，匹配的传入路由能够匹配添加了斜杠的URL。

提　示

如果在HTML响应中看到的URL是不同的格式，比如/?legacyurl=%2Farticles%2FWindows_3.1_Overview.html，那么说明自定义路由没有被用于生成URL，应用程序已经调用了其他的路由规则。由于没有指定控制器或操作，因此Home控制器上的Index操作方法将成为目标，legacyUrl值将被添加到URL查询字符串中。如果发生这种情况，要在 GetVirtualPath方法中将IsBound属性设置为true，检查Startup.cs文件中的配置是否为LegacyRoute构造函数指定了正确的URL，并且确认自定义路由定义在其他任何路由之前。

16.4 使用区域

ASP.NET Core MVC支持将Web应用程序组织

到各个区域（area），每个区域代表应用程序的功能部分，例如管理、计费、客户支持等。这在大型项目中非常有用，因为所有控制器、视图和模型都有一组文件夹可能会变得难以管理。

每个MVC区域都有自己的文件夹结构，允许保持一切独立分开。这就可以更明确哪些项目元素与应用程序的每个功能区域相关，可帮助多个开发人员在项目上进行工作而不会相互冲突。区域主要通过路由系统进行支持，这就是将此功能与URL和路由一起描述的原因。本节将介绍如何在MVC项目中设置和使用区域。

16.4.1　创建区域

为了创建区域，需要将一些文件夹添加到项目中。最上层的文件夹名为Areas，所有的区域都要包含在该文件夹中。每个区域都包含自己的

Controllers、Views和Models文件夹。在本章中，我们将创建一个名为Admin的区域，这意味着需要创建一组文件夹。为了准备示例项目，请创建表16-6中的所有文件夹。

表**16-6**　准备区域所需的文件夹

文件夹	描述
Areas	该文件夹将包含MVC应用程序中的所有区域
Areas/Admin	该文件夹将包含Admin区域的类和视图
Areas/Admin/Controllers	该文件夹将包含Admin区域的控制器
Areas/Admin/Views	该文件夹将包含Admin区域的视图
Areas/Admin/Views/Home	该文件夹将包含Admin区域中Home控制器的视图
Areas/Admin/Models	该文件夹将包含Admin区域的模型

虽然每个区域都是相互独立的，但许多MVC功能依赖于标准的C#或.NET功能，如命名空间。

为了让区域更容易使用，首先要在Views文件夹中引入文件，这样在视图中使用模型时就不需要再写命名空间名，并且可以使用标签助手。在Areas/Admin/Views文件夹中创建一个名为_ViewImports.cshtml的视图并引入文件，添加的语句如代码清单16-22所示。

代码清单16-22　Areas/Admin/Views文件夹下的_ViewImports.cshtml文件的内容

```
@using UrlsAndRoutes.Areas.Admin.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

16.4.2　创建区域路由

要使用区域，必须在Startup.cs文件中添加包含区域片段变量的路由，如代码清单16-23所示。

代码清单16-23　在UrlsAndRoutes文件夹下的Startup.cs文件中添加区域片段路由

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.Configure<RouteOptions>(options =>
 {
                options.ConstraintMap.Add("weekday", ty
peof(WeekDayConstraint));
                options.LowercaseUrls = true;
                options.AppendTrailingSlash = true;
            });
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "areas",
```

```
                    template: "{area:exists}/{controlle
r=Home}/{action=Index}");

                routes.Routes.Add(new LegacyRoute(
                    app.ApplicationServices,
                    "/articles/Windows_3.1_Overview.htm
l",
                    "/old/.NET_1.0_Class_Library"));

                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");

                routes.MapRoute(
                    name: "out",
                    template: "outbound/{controller=Hom
e}/{action=Index}");
            });
        }
    }
}
```

区域片段变量用于匹配特定区域中控制器的URL。在这里，按照标准的URL格式，可以设置成任何希望的格式。用于对区域添加支持的路由应该出现在不太具体的路由之前，以确保URL被正确匹配。exists约束用于确保请求仅与应用程序中定义的区域匹配。

16.4.3　填充区域

可以像在MVC应用程序的主要部分一样，在区域中创建控制器、视图和模型。要创建模型，可右击Areas/Admin/Models文件夹，从弹出的菜单中选择Add→Class并创建一个名为Person.cs的类文件，其中的内容如代码清单16-24所示。

代码清单16-24　Areas/Admin/Models文件夹下的Person.cs文件的内容

```
namespace UrlsAndRoutes.Areas.Admin.Models {
    public class Person {
        public string Name { get; set; }
        public string City { get; set; }
    }
}
```

要创建控制器，可右击Areas/Admin/Controllers文件夹，从弹出的菜单中选择Add→Class并创建一个名为HomeController.cs的类文件，用于定义控制器的内容，如代码清单16-25所示。

代码清单16-25　Areas/Admin/Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Areas.Admin.Models;

namespace UrlsAndRoutes.Areas.Admin.Controllers {

    [Area("Admin")]
    public class HomeController : Controller {
        private Person[] data = new Person[] {
            new Person { Name = "Alice", City = "London
" },
            new Person { Name = "Bob", City = "Paris" }
,
            new Person { Name = "Joe", City = "New York
" }
        };

        public ViewResult Index() => View(data);

    }
}
```

新的控制器是标准控制器，但和标准控制器不同的是，为了将控制器与区域相关联，必须在控制器上设置Area特性。

```
...
[Area("Admin")]
```

```
public class HomeController : Controller {
...
```

如果没有Area特性，控制器就不是区域的一部分，即使控制器是在应用程序的主要部分定义的也是如此。缺少Area特性可能导致奇怪的结果。在使用区域时，如果程序没有得到预期的结果，那么要做的第一件事就是检查是否正确添加了区域的Area特性。

提　示

如果使用特性设置路由（如第15章所述），则可以使用Route特性的参数中的[area]标记来引用Area特性指定的区域：

```
[Route("[area]/app/[controller]/actions/[action]/{id:weekday?}")]
```

在Areas/Admin/Views/Home文件夹中添加一个名为Index.cshtml的Razor视图，内容见代码清单16-26。

代码清单16-26　Areas/Admin/Views/Home文件夹下的Index.cshtml文件的内容

```
@model Person[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Areas</title>
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped table-sm">
        <tr><th>Name</th><th>City</th></tr>
```

```
        @foreach (Person p in Model) {
            <tr><td>@p.Name</td><td>@p.City</td></tr>
        }
    </table>
</body>
</html>
```

Index视图的模型是一个Person对象数组。由于在代码清单16-26中已经为视图引入文件，因此在这里可以直接使用Person类型而不需要再写命名空间。运行应用程序并请求URL /Admin以测试创建的区域，结果如图16-8所示。

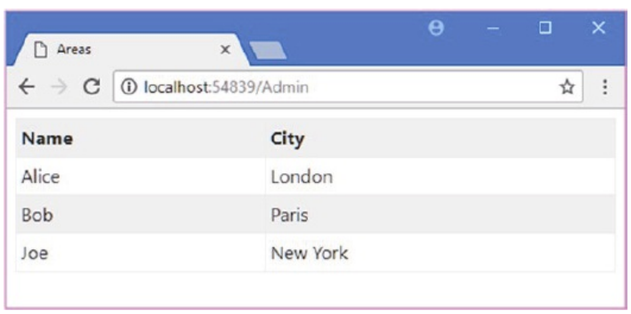


图16-8　使用区域

了解区域对MVC应用程序的影响

了解区域对其余应用程序的影响是非常重要的。这里创建了名为Admin的区域，但在应用程序中还有名为Admin的控制器。在创建区域之前，对/Admin URL的请求将被指向程序主目录中Admin控制器的Index操作。在添加区域后，请求将被指向Admin区域中Home控制器的Index操作（区域为controller和action片段变量提供了默认值）。这种更改可能会带来很多意外的情况，使用区域的最佳实践是将其用于项目的初始控制器命名方案。如果要在已经建立的应用程序中添加区域，就必须仔细考虑区域可能对路由带来的影响。

16.4.4　生成区域中指向操作的链接

不需要采取任何特殊步骤就能创建指向当前请求所在的同一MVC区域中操作的链接。MVC检测

到请求与特定区域相关，并确保传出的URL只能在为该区域定义的路由中找到匹配项。例如，在Areas/Admin/ Views/Home文件夹下的Index.cshtml文件中添加一个新的元素，如代码清单16-27所示。

代码清单16-27　在Areas/Admin/Views/Home文件夹下的Index.cshtml文件中添加锚点

```
@model Person[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Areas</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Name</th><th>City</th></tr>
        @foreach (Person p in Model) {
            <tr><td>@p.Name</td><td>@p.City</td></tr>
        }
```

```
    </table>
    <a asp-action="Index" asp-controller="Home">Link</a>

</body>
</html>
```

如果运行应用程序并请求URL /admin，你将看到响应包含以下元素：

```
<a href="/admin/">Link</a>
```

路由系统选择了用于生成传出URL的区域路由，并考虑了可用于controller和action片段变量的默认值。

必须为路由系统提供区域片段的值，以创建指向应用程序的不同区域或主要部分中的操作的链接，如代码清单16-28所示。

代码清单16-28　在Areas/Admin/Views/Home文件夹下的Index.cshtml文件中生成指向不同区域的链接

```
@model Person[]
```

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Areas</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped ta
ble-sm">
        <tr><th>Name</th><th>City</th></tr>
        @foreach (Person p in Model) {
            <tr><td>@p.Name</td><td>@p.City</td></tr>
        }
    </table>
    <a asp-action="Index" asp-controller="Home">Link</a
>
    <a asp-action="Index" asp-controller="Home" asp-rou
te-area="">Link</a>
</body>
</html>
```

asp-route-area属性用于设置区域片段变量的值。在这种情况下，将这个属性设置为空字符串，以指向应用程序的主要部分，并生成以下HTML元素：

```
<a href="/">Link</a>
```

如果控制器中有多个区域并且想要路由到它们，则需要使用区域名称代替上面的空字符串。

16.5 URL模式最佳实践

在介绍了上面这些内容后，你可能还不知道从哪里开始设计自己的URL模式。近年来，应用程序的URL设计越来越受到重视，一些重要的设计原则也应运而生。如果遵循这些设计模式，将能够提高应用程序的可用性、兼容性和搜索引擎排名。

16.5.1 保持URL的整洁性

下面是提高URL整洁性的一些建议。

- 设计的URL应该用来描述内容而不是体现应用程序实现的细节，比如应该使用/Articles/AnnualReport而不是

Website_v2/CachedContentServer/FromCache/Annu

- 尽量在URL中使用内容标题而不是数字ID，比如应尽量使用/Articles/AnnualReport而不是/Articles/2392。如果必须使用数字ID（以区分具有相同标题的项，或避免使用标题查找项所需的额外数据库查询），则建议使用/Articles/2392/AnnualReport。虽然这个URL看起来需要更长的录入时间，但是它对用户来说更有意义，并且有助于提高搜索引擎排名。应用可以直接忽略标题，利用 ID 来查找与之匹配的页面。
- 不要使用HTML页面的文件扩展名（例如.aspx或.mvc），但是要使用专门的文件类型（例如.jpg、.pdf和.zip）。必须正确设置MIME类型，Web浏览器不关心文件扩展名，但人们仍然希望PDF文件以.pdf结尾。
- 创建的URL要具有层次感（例如/Products/Menswear/Shirts/Red），以便访问者可以猜测父类别的URL。
- 不区分大小写。默认情况下，ASP.NET Core路

由系统不区分大小写。

- 避免使用符号、代码和字符序列。如果需要单词分隔符，可使用连字符（比如/my-great-article）。下画线并不友好。在URL编码中，空格的编码很奇怪（/my+great+article），也让人难受（/my%20great%20article）。
- 不要更改URL。无效的链接会导致丢失业务。当更改URL时，可以通过重定向继续支持旧的URL模式。
- 保持一致。在整个应用程序中采用一种URL格式。

网址应该很简单，易于输入，并且人性化（可编辑）且具有持久性，还应该可视化网站结构。URL可用性专家Jakob Nielsen扩展了这个主题。Web专家Tim Berners-Lee也提供了类似的建议。

16.5.2　GET方法和POST方法：选择最合适的方法

经验法则是，将GET请求应用于所有只读信息的检索，而将POST请求应用于任何更改应用程序状态的操作。在符合标准的条款中，GET请求是为了安全交互（除信息检索以外没有副作用），而POST请求则用于不安全的交互（做出决定或更改某些内容）。这些约定是由万维网联盟（W3C）制定的。

GET请求是可寻址的，所有的信息都包含在URL中，因此可以将这些网址收藏为书签或创建链接。

需要在更改状态时使用GET方法，当Google Web Accelerator在2005年向公众发布时，很多网站开发人员有过惨痛的教训。Google Web Accelerator收录每个页面的所有链接，这在HTTP中是合法的，因为GET请求应该是安全的。但是，许多Web开发人员忽略了HTTP约定，并在应用程序中使用

了“删除商品”或“添加到购物车”的链接，混乱也就随之而来。

一家公司认为自己的管理系统正在遭受攻击，因为所有的内容都在不断地被删除，后来该公司才发现原来是搜索引擎抓取工具抓取了管理页面的URL，并且在抓取所有删除数据的链接。如果有身份验证，就可以保护你不受此影响，但无法让你免受搜索引擎爬虫的困扰。

16.6 小结

本章展示了路由系统的高级功能，如何生成出站链接和URL，以及如何自定义路由系统。此外，本章还介绍了区域的概念，并讨论了如何创建有效且有意义的URL模式。下一章将介绍控制器和操作，这是ASP.NET Core MVC的核心内容，除详细解释这些内容之外，还将展示如何使用它们在应用

中获得最佳效果。

第17章　控制器和操作

每个应用程序的请求都由控制器处理。在ASP.NET Core MVC中，控制器是包含处理请求所需逻辑的.NET类。第3章介绍过控制器的作用是封装应用程序逻辑。这意味着控制器负责处理传入的请求，对模型执行操作，并选择要呈现给用户的视图。

只要不偏离属于模型和视图的责任区域，控制器就可以自由地处理自认为合适的任何方式的请求。这意味着控制器既不包含或存储数据，也不生成用户界面。

本章将展示如何实现控制器以及使用控制器接收和生成数据的不同方式。表17-1展示了在上下文中使用的控制器。

表17-1　在上下文中使用的控制器

问题	答案
什么是控制器？	控制器包含用于接收请求、更新应用程序状态或模型以及选择将发送给客户端的响应的逻辑
控制器有什么用？	控制器是MVC项目的核心，并包含Web应用程序的逻辑
如何使用控制器？	控制器是调用公共方法来处理HTTP请求的C#类。这些方法可以负责直接为客户生成响应，但更常见的方式是返回操作结果，告诉MVC应该如何去响应
使用控制器是否有一些隐患或局限性？	刚开始使用MVC时，可以轻松创建包含更适合于模型或视图功能的控制器。一个更为具体的问题是任何以Controller为名的公共类都会被假定为MVC的控制器，这意味着一些可能不是作为控制器的类被意外用于处理HTTP请求
是否有其他的替代选择？	没有，控制器是MVC应用程序的核心部分

表17-2总结了本章要介绍的操作。

表17-2　本章要介绍的操作

操作	方法	代码清单

定义控制器	创建一个公共类，名称以Controller为结尾或派生自Controller类	代码清单17-7～代码清单17-9
获取HTTP请求的详细信息	使用context对象或定义操作方法参数	代码清单17-10～代码清单17-13
从操作方法生成结果	直接使用context结果对象或创建操作结果对象	代码清单17-14～代码清单17-16
生成HTML结果	创建视图结果	代码清单17-17～代码清单17-24
重定向客户端	创建重定向结果	代码清单17-25～代码清单17-30
向客户端返回内容	创建内容结果	代码清单17-31～代码清单17-35
返回HTTP状态码	创建HTTP结果	代码清单17-36和代码清单17-37

17.1　准备示例项目

在本章中，将使用ASP.NET Core Web Application（.NET Core）模板创建一个名为

ControllersAndActions的新的Empty项目。代码清单17-1在Startup类中添加了一些语句，以启用MVC框架和其他中间件。

注　意

本章包括用于关键功能的单元测试。为简洁起见，没有将单元测试项目包含在创建示例项目的说明中。可以按照第7章中描述的过程创建测试项目，或从本书的GitHub存储库下载该项目。

代码清单17-1　在ControllersAndActions文件夹下的Startup.cs文件中添加MVC和其他中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ControllersAndActions {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

AddMemoryCache和AddSession方法将创建会话管理所需的服务。UseSession方法用于将一个中

间件添加到管道中，将会话数据与请求关联起来，并将cookie添加到响应中，以确保将来的请求能够被识别。必须在UseMvc方法之前调用UseSession方法，以便会话组件能够在到达MVC中间件之前截获请求，并且可以在生成后修改响应。其他几个方法用于设置第14章中描述的标准包。

准备视图

本章的重点是控制器及其操作方法，本章将定义控制器类。这里将定义一些视图，它们可以帮助展示控制器类的工作方式。这些视图创建在Views/Shared文件夹中，以便可以在本章后面创建的任何控制器中使用它们。创建Views/Shared文件夹，添加一个名为Result.cshtml的Razor视图文件，内容如代码清单17-2所示。

代码清单17-2　Views/Shared文件夹下的Result.cshtml文件中的

内容

```
@model string
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    Model Data: @Model
</body>
</html>
```

用于Result视图的模型是一个字符串，这将允许显示简单的消息。接下来，在Views/Shared文件夹中创建一个名为DictionaryResult.cshtml的文件，内容如代码清单17-3所示，用于DictionaryResult视图的模型是一个字典，用于显示比前一个视图更复杂的数据。

代码清单17-3　Views/Shared文件夹下的

DictionaryResult.cshtml文件的内容

```
@model IDictionary<string, string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-sm table-s
triped">
        <tr><th>Name</th><th>Value</th></tr>
        @foreach (string key in Model.Keys) {
            <tr><td>@key</td><td>@Model[key]</td></tr>
        }
    </table>
</body>
</html>
```

接下来，在Views/Shared文件夹中创建一个名为SimpleForm.cshtml的文件，内容如代码清单17-4所示。顾名思义，SimpleForm视图包含一个简单的HTML表单，用于提交数据。

代码清单17-4　Views/Shared文件夹下的SimpleForm.cshtml文件的内容

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <form method="post" asp-action="ReceiveForm">
        <div class="form-group">
            <label for="name">Name:</label>
            <input class="form-control" name="name" />
        </div>
        <div class="form-group">
            <label for="name">City:</label>
            <input class="form-control" name="city" />
        </div>
        <button class="btn btn-primary center-block" ty
pe="submit">Submit</button>
    </form>
</body>
</html>
```

这些视图使用内置的标签助手从路由系统生成URL。要启用标签助手，可在Views文件夹中创建

一个名为_ViewImports.cshtml的视图导入文件，内容如代码清单17-5所示。

代码清单17-5　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

在Views/Shared文件夹中创建的视图都依赖于Bootstrap CSS包。要将Bootstrap添加到项目中，可使用Bower Configuration File模板创建bower.json文件，添加的内容如代码清单17-6所示。

代码清单17-6　在ControllersAndActions文件夹下的bower.json文件中添加包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

17.2 理解控制器

控制器是C#类，其公共方法（称为action或操作方法）负责处理HTTP请求并准备将返回给客户端的响应。MVC使用第15章和第16章描述的路由系统来确定处理请求所需的控制器类和操作方法。然后，MVC创建一个控制器类的新实例，调用操作方法，并使用调用结果来向客户端输出响应。

MVC提供了上下文数据的操作方法，以便它们能够理解如何处理请求。有大量的上下文数据，它们描述了当前请求的所有内容、正在准备的响应、由路由系统提取的数据以及用户身份的详细信息。

当MVC调用操作方法时，操作方法的响应描述了应该发送给客户端的响应。最常见的响应是创建并渲染Razor视图，所以操作方法使用响应来告

诉MVC要使用哪个视图，以及应该提供哪些视图模型数据。此外还能提供其他类型的响应，并且操作方法可以从MVC中发送HTTP重定向到客户端以发送复杂的数据对象。

这意味着可从3个重要的方面来理解控制器。首先，了解如何定义控制器，以便MVC可以使用它们来处理请求。控制器只是C#类，但和一般类的创建方法不同，了解它们之间的差异很重要。17.3节将解释如何定义控制器。

其次，了解MVC如何提供上下文数据的操作方法也是很重要的。获取所需的上下文数据对于有效的Web应用开发非常重要，通过定义一组用于描述操作方法所需的所有内容的类，MVC可以使Web应用开发变得容易。17.4节将解释MVC如何描述请求和响应。

最后，了解操作方法如何产生响应也很重要。操作方法很少需要自己生成HTTP响应，你需要知道如何使用MVC生成所需的响应，17.5节将对此进行解释。

17.3　创建控制器

到目前为止，几乎所有章节都使用了控制器。现在是时候回头来看看它们是如何定义的。本节将描述控制器的不同创建方式，并解释它们之间的差异。

17.3.1　创建POCO控制器

MVC配置十分方便灵活，这意味着MVC应用程序中的控制器是自动发现的，而不用在配置文件中定义。基本的发现过程很简单：名称以Controller结尾的任何公共类都是控制器，其中定义的任何公

共方法都是操作方法。为了演示这是如何工作的，在项目中添加Controllers文件夹，并在其中添加一个名为PocoController.cs的类文件，内容如代码清单17-7所示。

提　示

虽然约定是将控制器放在Controllers文件夹中，但是可以将它们放在项目的任何位置，MVC仍然会自动找到它们。

代码清单17-7　Controllers文件夹下的PocoController.cs文件的内容

```
namespace ControllersAndActions.Controllers {

    public class PocoController {

        public string Index() => "This is a POCO contro
ller";
    }
}
```

PocoController类满足了MVC控制器的基础标准，它定义了一个名为Index的公共方法，可作为一个操作方法并返回一个字符串。

PocoController类是POCO控制器的范例，其中POCO表示“普通的旧CLR对象”，并且指的是使用标准.NET功能实现控制器，而不直接依赖于ASP.NET Core MVC提供的API。

要测试POCO控制器，可启动应用程序并请求URL /Poco/Index/。路由系统将使用默认URL模式匹配请求，并将请求指向PocoController类的Index方法，结果如图17-1所示。



图17-1　使用POCO控制器

使用属性标识调整控制器

对POCO控制器的支持并不总是按希望的方式工作。一个常见的问题是，MVC将以单元测试中的假类作为控制器。避免这个问题的最简单方法是注重类名，避免名字类似于FakeController。如果无法避免，那么可以使用命名空间Microsoft.AspNetCore.Mvc中定义的NonController属性，用于告诉MVC这个类不是控制器类。NonAction属性可应用于方法，以阻止它们被作为操作方法使用。

在某些项目中，可能无法在作为POCO控制器的类上遵循命名约定。通过应用Controller属性（在命名空间Microsoft.AspNetCore.Mvc中定义），可以告诉MVC这是控制器类，即使不符合POCO选择标准。

使用**MVC API**

PocoController类是MVC识别控制器以及简单使用控制器的有用演示。但是，不依赖Microsoft.AspnetCore命名空间的纯POCO控制器并不是特别有用，因为它们无法访问MVC为处理请求提供的功能。

可以通过从Microsoft. AspnetCore命名空间创建新的实例来访问MVC API的某些功能。作为一个简

单的例子，POCO类可以通过从操作方法返回一个ViewResult对象来使MVC呈现Razor视图，如代码清单17-8所示（详见17.5节的ViewResult类）。

代码清单17-8　在Controllers文件夹下的PocoController.cs文件中使用ASP.NET API

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
namespace ControllersAndActions.Controllers {

    public class PocoController {

        public ViewResult Index() => new ViewResult() {
                ViewName = "Result",
                ViewData = new ViewDataDictionary(
                    new EmptyModelMetadataProvider(),
                    new ModelStateDictionary()) {
                        Model = $"This is a POCO contro
ller"
                    }
            };
    }
}
```

这不再是纯POCO控制器，因为它直接依赖于MVC API。除纯粹之外，它比前一个例子有用得

多，因为它要求MVC渲染一个Razor视图。

但是，代码很复杂。为了创建一个ViewResult对象，需要创建ViewDataDictionary、EmptyModelMetadataProvider和ModelStateDictionary对象，这需要访问3个不同的命名空间（后面的章节将描述这些类型所涉及的特性）。此例的重点是说明MVC提供的功能可以直接访问，即使结果有点混乱。

我们使用一个字符串作为视图模型来渲染Result.cshtml，如果你运行应用程序并请求URL /Poco/Index，你看到的结果将如图17-2所示。

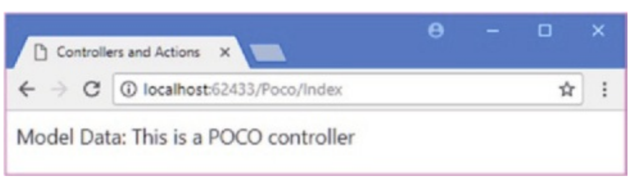


图17-2　直接使用MVC API请求的结果

17.3.2　使用控制器基类

之前的例子展示了如何从POCO控制器开始，并对其进行扩展来访问MVC功能。这种方法揭示了MVC的工作原理，这是很有用的知识，但是这样会使POCO控制器难以编写、阅读和维护。

创建控制器的一种更简单的方法是从Microsoft.AspNetCore.Mvc.Controller类派生类，基类定义了一些方法和属性，从而以更简洁有用的方式访问MVC功能。为了演示，将一个名为DerivedController.cs的类文件添加到Controllers文件夹中，内容如代码清单17-9所示。

代码清单17-9　在Controllers文件夹下的由Controller类派生出的DerivedController.cs文件

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class DerivedController : Controller {
```

```
        public ViewResult Index() =>
            View("Result", $"This is a derived controll
er");
    }
}
```

如果运行应用程序并访问/Derived/Index，你看到的结果将如图17-3所示。

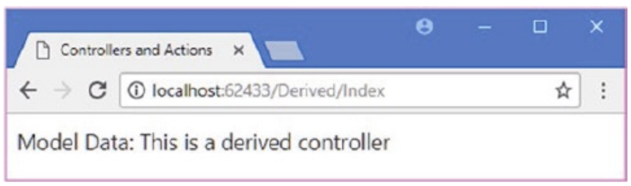


图17-3　使用Controller基类

代码清单17-9中的控制器执行与代码清单17-8中的控制器相同的操作（要求MVC渲染具有字符串视图模型的视图），但是使用Controller基类意味着可以更简单地实现结果。

关键的变化是，可以使用View方法创建渲染Razor视图所需的ViewResult对象，而不必直接在操作方法中进行实例化。View方法继承自Controller基

类，ViewResult对象仍然以相同的方式创建，只是没有多余代码来扰乱操作方法。从Controller类进行派生不会改变控制器的工作方式，而只是简化了编写的代码以完成常见任务。

注　意

MVC会为要求处理的每个请求创建一个控制器类的新实例，这意味着不需要同步对操作方法或实例属性和字段的访问。第18章将要描述的共享对象（包括数据库和单例服务）可以同时使用，并且必须根据规则来编写。

17.4　接收上下文数据

无论如何定义控制器，它们几乎都不会孤立存在，通常需要从传入的请求中访问数据，例如查询字符串值、表单值以及路由系统从URL中解析的参数，统称为上下文数据。访问上下文数据有3种主要方式：

- 从一组Context对象中提取。
- 接收数据作为操作方法的参数。
- 明确地调用框架的模型绑定功能。

这里将介绍如何在操作方法获取输入，重点是使用Context对象和操作方法参数。第26章将介绍模型绑定。

17.4.1　从Context对象中接收数据

使用Controller基类创建控制器的主要优点之一

是可以方便地访问描述当前请求、正在准备的响应和应用程序状态的一组Context对象。表17-3描述了用于上下文数据的一些有用的控制器类属性。

表**17-3**　用于上下文数据的一些有用的控制器类属性

属性	描述
Request	返回一个HttpRequest对象，用于描述从客户端收到的请求
Response	返回用于创建客户端响应的HttpResponse对象
HttpContext	返回一个HttpContext对象，它是由其他属性（如请求和响应）返回的许多对象的源，它还提供了有关可用的HTTP功能的信息以及对低级功能（如Web套接字）的访问
RouteData	返回由路由系统在匹配请求时生成的RouteData对象
ModelState	返回一个ModelStateDictionary对象，用于验证客户端发送的数据
User	返回一个ClaimsPrincipal对象，用于描述已发出请求的用户

许多控制器的编写不需要使用表17-3中的属性，因为上下文数据也可以通过后面几章描述的功

能提供，这更符合MVC开发风格。例如，大多数控制器不需要使用Request属性来获取正在处理的HTTP请求的详细信息，因为通过第26章描述的模型绑定过程可以获得相同的数据。

但是，理解和使用Context对象仍然有用，而且它们对于调试很有用。代码清单17-10使用Request属性来访问HTTP请求中的标头。

代码清单17-10　在Controllers文件夹下的DerivedController.cs文件中使用上下文数据

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace ControllersAndActions.Controllers {

    public class DerivedController : Controller {

        public ViewResult Index() =>
            View("Result", $"This is a derived controll
er");

        public ViewResult Headers() => View("Dictionary
Result",
                Request.Headers.ToDictionary(kvp => kvp
```

```
.Key,
                                kvp => kvp.Value.First()));
    }
}
```

使用Context对象意味着浏览一系列不同的类型和命名空间。用来获取有关列表中HTTP请求的上下文数据的Controller.Request属性会返回一个HttpRequest对象。表17-4描述了在编写控制器时有用的HttpRequest属性。

表**17-4**　有用的**HttpRequest**属性

属性	描述
Path	返回请求URL的路径部分
QueryString	返回请求URL的查询字符串部分
Headers	返回按名称索引的请求标头的字典
Body	返回可用于读取请求正文的流
Form	返回请求中表单数据的字典（按名称索引）

Cookies	返回按名称索引的请求cookie的字典

可使用Request.Headers来获取标头的字典，并使用LINQ对其进行处理。

```
...
View("DictionaryResult", Request.Headers.ToDictionary(kvp => kvp.Key,
      kvp => kvp.Value.First()));
...
```

由Request.Headers属性返回的字典使用StringValues结构存储每个标头的值，该结构在ASP.NET中用于表示字符串值序列。HTTP客户端可以为HTTP标头发送多个值，但是这里只想显示第一个值。可使用LINQ ToDictionary方法为每个标题接收一个KeyValuePair<String,StringValues>对象，并选择第一个值。结果是包含字符串值的字典，可以由DictionaryResult视图显示。如果运行应用程序并请求URL /Derived/Headers，你将看到的

输出类似于图17-4（根据使用的浏览器，标头及值将有所不同）。

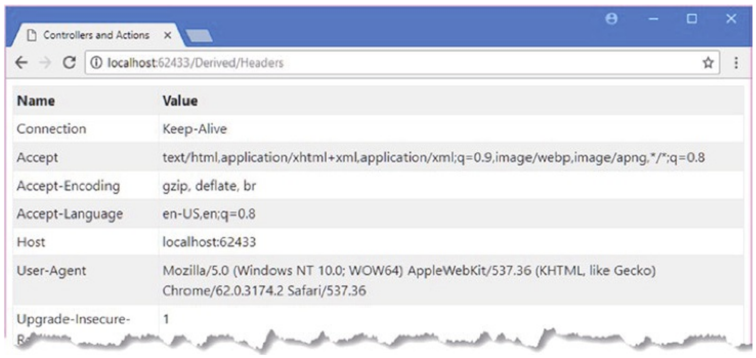


图17-4　显示上下文数据

在**POCO**控制器中获取上下文数据

即使它们在常规项目中不是特别有用，POCO控制器也能让我们在幕后查看MVC是如何操作的。在POCO控制器中获取上下文数据是一个问题，因为不能只实例化自己的HttpRequest或HttpResponse对象，还需要那些已由ASP.NET创建并由所有在处理请求时填充了数据字段的中间件更新的对象。

要获取上下文数据，POCO控制器必须要求

MVC提供在前。代码清单17-11更新了PocoController类以添加显示HTTP请求标头的操作方法。

代码清单17-11　在Controllers文件夹下的PocoController.cs文件中显示上下文数据

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System.Linq;

namespace ControllersAndActions.Controllers {

    public class PocoController {

        [ControllerContext]
        public ControllerContext ControllerContext { ge
t; set; }

        public ViewResult Index() => new ViewResult() {
            ViewName = "Result",
            ViewData = new ViewDataDictionary(new Empty
ModelMetadataProvider(),
                            new ModelStateDictionary())
{
                Model = $"This is a POCO controller"
            }
        };
        public ViewResult Headers() =>
```

```
            new ViewResult() {
                ViewName = "DictionaryResult",
                ViewData = new ViewDataDictionary(
                    new EmptyModelMetadataProvider(),
                    new ModelStateDictionary()) {
                        Model = ControllerContext.HttpC
ontext.Request.Headers
                        .ToDictionary(kvp => kvp.Key, k
vp => kvp.Value.First())
                }
            };
    }
}
```

为了获取上下文数据，可定义一个名为ControllerContext的属性，类型为ControllerContext，并使用同名的ControllerContext属性进行装饰。

下面介绍3种不同的ControllerContext的用途。首先，Microsoft.AspNetCore.Mvc命名空间中定义的ControllerContext类用于使用ControllerContext的属性（见表17-5）汇集控制器的操作方法所需的所有Context对象。

表**17-5　ControllerContext**的属性

属性	描述
ActionDescriptor	返回一个ActionDescriptor对象，用于描述操作方法
HttpContext	返回一个HttpContext对象，用于提供HTTP请求的详细信息和发送回客户端的HTTP响应
ModelState	返回一个ModelStateDictionary对象，用于验证客户端发送的数据
RouteData	返回一个RouteData对象，用于描述路由系统处理请求的方式

通过ControllerContext.HttpContext属性可访问HTTP相关数据，该属性会返回一个Microsoft.AspNetCore. Http.HttpContext对象。HttpContext类合并了几个用于描述请求的不同方面的对象，可通过HttpContext的属性（见表17-6）进行访问。

表**17-6　HttpContext**的属性

属性	描述
Connection	返回一个ConnectionInfo对象，用于描述与客户端的低级别连接
Request	返回一个HttpRequest对象，用于描述从客户端接收的HTTP请求
Response	返回一个HttpResponse对象，用于创建将返回给客户端的响应
Session	返回一个用于描述与请求关联的会话的ISession对象
User	返回一个ClaimPrincipal对象，用于描述与请求关联的用户

其次，ControllerContext属性用于描述代码清单17-11中的属性，并告诉MVC使用描述当前请求的ControllerContext对象设置属性值。这使用了一种称为依赖注入的技术，这将在第18章中介绍，MVC将使用该属性，在使用操作方法处理请求之前为控制器提供上下文数据。

最后，ControllerContext是属性的名称。可以在自己的POCO控制器中使用任何合法的C#属性名称，在幕后，Controller类依赖于相同的

ControllerContext类的上下文数据，该类使用相同的ControllerContext属性进行装饰。表17-3中描述的所有控制器属性都是直接使用ControllerContext属性的更方便、更简洁的替代方法，这正是控制器类提供的属性的使用情况。例如，下面是控制器类中的HttpContext属性的定义：

```
...
public HttpContext HttpContext {
    get {
        return ControllerContext.HttpContext;
    }
}
...
```

HttpContext属性只是获取ControllerContext.HttpContext属性值的一种手段。控制器基类没有魔法：它们只不过是更简单、更清晰的控制器，因为常见任务已被整合到方法和属性中。如有需要，可以在POCO控制器中重新创建自己所需的内容。ASP.NET Core MVC中的很多功能都是非常简单

的，深入挖掘细节，你将发现并没有特别的内容，只有精心设计的功能，并且提供了一套详细的NuGet包。如果有时间，建议从GitHub下载MVC源代码并进行浏览，从而确认这一点。

17.4.2　使用操作方法参数

一些上下文数据也可以通过操作方法参数来接收，这可以让我们编写更自然且优雅的代码。一个常见的例子是当操作方法需要接收用户提交的表单数据时。为了比较，下面演示如何通过Context对象获取表单数据，然后通过操作方法获取参数。

表单数据可通过Controller类的Request.Form属性访问。为了演示，添加一个名为HomeController.cs的类文件，并用它定义派生控制器，如代码清单17-12所示。

代码清单17-12　Controllers文件夹下的HomeController.cs文件

的内容

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        public ViewResult ReceiveForm() {
            var name = Request.Form["name"];
            var city = Request.Form["city"];
            return View("Result", $"{name} lives in {ci
ty}");
        }
    }
}
```

以上控制器中的Index操作方法将呈现本章开头在Views/Shared中创建的SimpleForm视图。我们需要关注ReceiveForm方法，以使用HttpRequest context对象从请求中获取表单数据。

如表17-4所示，由HttpRequest类定义的Form属性将返回一个包含表单数据的集合，并由关联的

HTML元素的名称进行索引。SimpleForm视图中有两个输入元素（name和city），可从Context对象中提取它们的值，并使用它们创建一个字符串作为模型传递给Result视图。

如果运行应用程序并请求URL——/Home，将会显示一个表单。如果填写字段并单击Submit按钮，浏览器将发送表单数据作为HTTP POST请求的一部分，由ReceiveForm方法处理，产生的结果如图17-5所示。

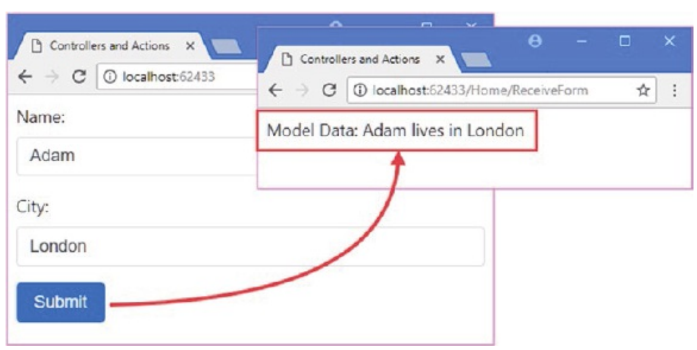


图17-5　从Context对象获取表单数据

代码清单17-12所示的方法很有效，但是还有

更优雅的选择。操作方法可以定义MVC使用的参数，将上下文数据传递给控制器，包括HTTP请求的详细信息。这比直接从Context对象提取更简单，并且能产生更容易阅读的操作方法。要接收表单数据，请在名称对应的表单数据的操作方法上声明参数，如代码清单 17-13所示。

代码清单17-13　在Controllers文件夹下的HomeController.cs文件中以参数方式接收上下文数据

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        public ViewResult ReceiveForm(string name, stri
ng city)
            => View("Result", $"{name} lives in {city}"
);
    }
}
```

修改后的操作方法会产生相同的结果，但更易于阅读和理解。MVC将通过自动检查Context对象（包括请求字符串、表单和RouteData的值）来提供操作方法参数的值。参数的名称分情况处理，以便可以通过Request. Form["City"]中的值填充名为city的操作方法参数。以上方式还会生成更容易进行单元测试的操作方法，因为操作方法操作的值可作为常规的C#参数接收，并且不需要Context对象。

17.5　生成响应

操作方法在处理完请求之后，需要生成响应。有许多功能可用于从操作方法生成输出，下面进行详细介绍。

17.5.1　使用Context对象生成响应

生成输出的最简单方法是使用HttpResponse

Context对象，该对象指定了ASP.NET Core如何访问发送到客户端的HTTP响应。表17-7描述了由HttpResponse类提供的基本属性，该类定义在Microsoft.AspNetCore.Http命名空间中。

表**17-7　HttpResponse**提供的基本属性

属性	描述
StatusCode	用于设置响应的HTTP状态码
ContentType	用于设置响应的Content-Type标头
Headers	返回将包含在响应中的HTTP标头的字典
Cookies	返回一个用于向响应添加Cookies的集合
Body	返回用于写入响应的主体数据的System.IO.Stream对象

代码清单17-14更新了Home控制器，从而在ReceivedForm操作方法中使用Controller.Request属性返回HttpResponse对象来生成响应。

代码清单17-14　在Controllers文件夹下的HomeController.cs文件中生成响应

```
using Microsoft.AspNetCore.Mvc;
using System.Text;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        public void ReceiveForm(string name, string cit
y) {
            Response.StatusCode = 200;
            Response.ContentType = "text/html";
            byte[] content = Encoding.ASCII
                .GetBytes($"<html><body>{name} lives in
 {city}</body>");
            Response.Body.WriteAsync(content, 0, conten
t.Length);
        }
    }
}
```

但这种生成响应的方式是很可怕的，因为它使用C#字符串对操作方法中的HTML进行硬编码，这容易出错且难以进行单元测试。但这种方式确实有

助于了解如何在幕后创建响应。

还有比直接使用HttpResponse对象更好的替代方法。MVC在底层响应的基础上提供了一个更有用的功能，并且也是控制器工作的核心——操作结果。

17.5.2 理解操作结果

MVC使用操作结果来分离说明意图和执行意图。这个概念很简单，但是首先需要一些时间才能理解。

操作方法将返回一个从Microsoft.AspNetCore.Mvc命名空间实现了IActionResult接口的对象，而不是直接使用HttpResponse对象。IActionResult对象又称为操作结果，它描述了控制器应该如何响应，例如呈现视图或将客户端重定向到另一个URL。但是，它不会直

接生成响应，而是通过MVC的操作结果来产生结果。

注　意

操作结果系统是命令模式的一个例子。命令模式描述了一种场景，以存储和传递这样的对象——这种对象描述了要执行的操作。

下面是来自MVC源代码的IActionResult接口的定义：

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc {
```

```
    public interface IActionResult {
        Task ExecuteResultAsync(ActionContext context);
    }
}
```

这个接口可能看起来很简单，但这是因为MVC不会规定操作结果可以产生什么样的响应。当操作方法返回操作结果时，MVC将调用ExecuteResultAsync方法，该方法负责生成响应以代表操作方法。ActionContext参数提供了用于生成响应的上下文数据，包括HttpResponse对象（ActionContext类是ControllerContext的超类，并定义了表17-5中描述的所有属性）。

为了演示操作结果如何工作，为项目添加Infrastructure文件夹，并添加一个名为CustomHtmlResult.cs的类文件，内容如代码清单17-15所示。

代码清单17-15　Infrastructure文件夹下的CustomHtmlResult.cs

文件的内容

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using System.Threading.Tasks;

namespace ControllersAndActions.Infrastructure {

    public class CustomHtmlResult : IActionResult {

        public string Content { get; set; }
        public Task ExecuteResultAsync(ActionContext co
ntext) {
            context.HttpContext.Response.StatusCode = 2
00;
            context.HttpContext.Response.ContentType =
"text/html";
            byte[] content = Encoding.ASCII.GetBytes(Co
ntent);
            return context.HttpContext.Response.Body.Wr
iteAsync(content,
                0, content.Length);
        }
    }
}
```

CustomHtmlResult类实现了IActionResult接口，其ExecuteResultAsync方法使用HttpResponse对象来编写HTML响应，其中包含Content属性的值。ExecuteResultAsync方法必须返回一个Task对象，以

便可以异步生成响应；这适用于CustomHtmlResult类中的实现，该类依赖于表示响应体的Stream对象的WriteAsync方法，并返回可以用作操作结果的Task方法。

在代码清单17-16中，已通过在Home控制器中使用操作结果简化了Home控制器的ReceiveForm操作方法。

代码清单17-16　在Controllers文件夹下的HomeController.cs文件中使用操作结果

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        public IActionResult ReceiveForm(string name, s
tring city)
            => new CustomHtmlResult {
```

```
                Content = $"{name} lives in {city}"
            };
        }
}
```

发送响应的代码现在已与响应包含的数据分开定义，这简化了操作方法，并允许在其他操作方法中生成相同类型的响应，而不会产生重复的代码。

用于控制器和操作的单元测试

ASP.NET Core MVC的许多部分是为了方便单元测试而设计的，对于操作和控制器来说尤其如此。ASP.NET Core MVC支持单元测试，具体表现在以下几个方面。

- 可以在Web服务器之外测试操作和控制器。
- 不需要解析任何HTML来测试操作方法的结果。可以检查返回的IActionResult对象，以确保收到预期的结果。

- 不需要模拟客户端请求。MVC模型绑定系统允许编写作为方法参数接收输入的操作方法。为了测试操作方法，只需要直接调用操作方法，并提供希望的参数值即可。

本章将介绍如何为不同类型的操作结果创建单元测试。有关设置单元测试项目的说明，请参阅第7章，或从GitHub下载示例项目。

17.5.3　生成HTML响应

刚才介绍了如何使用操作结果从控制器类中获取生成响应的代码。ASP.NET Core MVC提供了更灵活的方法来生成响应——使用ViewResult类。

ViewResult类是能够提供对Razor视图引擎的访问的操作结果，该引擎处理.cshtml文件以合并模型数据，并通过HttpResponse上下文引擎将结果发送

给客户端。第21章将解释视图引擎的工作原理，在本章中，重点是使用ViewResult类作为操作结果。

在代码清单17-17中，已经用ViewResult替换了自定义的操作结果类，操作结果类可通过Controller基类提供的View方法创建。

代码清单17-17　使用Controllers文件夹下的HomeController.cs文件中的ViewResult类

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        public ViewResult ReceiveForm(string name, stri
ng city)
            => View("Result", $"{name} lives in {city}"
);
    }
}
```

可以直接创建ViewResult对象，正如本章开头的POCO控制器中演示的那样，但是使用View方法更简单、更简洁。Controller类提供了几个不同版本的View方法，允许选择渲染的视图并提供模型数据，如表17-8所示。

表**17-8　Controller**类中的**View**方法

View方法	描述
View()	为与操作方法关联的默认视图创建一个ViewResult对象，以便在名为MyAction的方法中调用View()，呈现名为MyAction.cshtml的视图。不使用模型数据
View(view)	创建一个将呈现指定视图的ViewResult对象，调用View（"MyView"）将呈现一个名为MyView.cshtml的视图。不使用模型数据
View(model)	为与操作方法关联的默认视图创建一个ViewResult对象，并使用指定对象作为模型数据
View(view,model)	为指定的视图创建一个ViewResult对象，并使用指定对象作为模型数据

如果运行应用程序并提交表单，你看到的结果

将如图17-6所示。

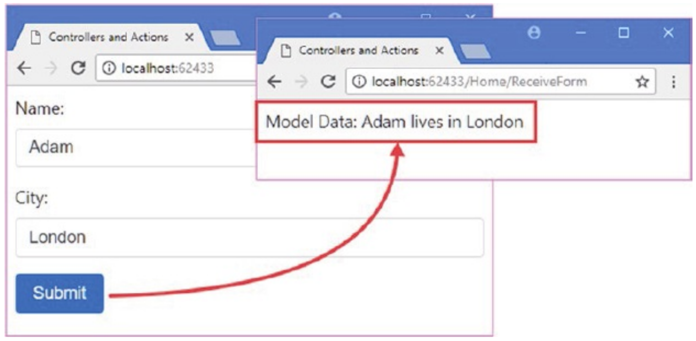


图17-6　使用ViewResult对象生成HTML响应结果

1．理解对视图文件的搜索

当MVC调用ViewResult对象的ExecuteResult-Async方法时，将从指定的视图开始搜索。MVC搜索视图的目录顺序是与配置相关的约定示例。不需要使用框架注册视图文件。只需要将它们放到一组已知位置，框架即可找到它们。默认情况下，MVC将在以下位置查找视图：

```
/Views/<ControllerName>/<ViewName>.cshtml
/Views/Shared/<ViewName>.cshtml
```

搜索从包含专用于当前控制器的视图的文件夹开始。该文件夹的名称中省略了类名的Controller部分，以便HomeController类的文件夹名为Views/Home。

如果视图名未在ViewResult对象中指定，就使用路由数据中action变量的值。对于大多数控制器来说，这意味着将使用方法的名称，以使与Index方法关联的默认视图文件为Index.cshtml。但是，如果已使用Route属性，那么与操作方法关联的视图名可能不同。

如果控制器是区域的一部分，如第16章所述，则搜索位置也会不同。

```
/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml
/Areas/<AreaName>/Views/Shared/<ViewName>.cshtml
/Views/Shared/<ViewName>.cshtml
```

MVC依次检查这些文件是否存在。一旦找到匹配，就会使用视图呈现操作方法的结果。这里因为没有在示例项目中使用区域，所以代码清单17-18中的操作方法会使MVC通过查找Views/Home/Result.cshtml文件来开始搜索。没有这样的文件，所以搜索继续，MVC寻找Views/Shared/Result.cshtml，它确实存在，因此它用于呈现HTML响应。

单元测试——渲染视图

要测试操作方法呈现的视图，可以检查返回的ViewResult对象。这和检查输出的视图并不完全相同（毕竟，你没有按照过程检查生成的最终HTML），但只要MVC视图系统正常工作就足够了。为项目添加一个名为ActionTests.cs的单元测试

文件，以保存本章执行的单元测试。想要测试的第一种情况是当操作方法选择特定的视图时，如下所示：

```
...
public ViewResult ReceiveForm(string name, string city)
    => View("Result", $"{name} lives in {city}");
...
```

可以通过读取ViewResult对象的ViewName属性来确定已选择哪个视图，如下所示：

```
using ControllersAndActions.Controllers;
using Microsoft.AspNetCore.Mvc;
using Xunit;

namespace ControllersAndActions.Tests {

    public class ActionTests {

        [Fact]
        public void ViewSelected() {
            // Arrange
            HomeController controller = new HomeControl
ler();

            // Act
            ViewResult result = controller.ReceiveForm(
"Adam", "London");
```

```
            // Assert
            Assert.Equal("Result", result.ViewName);
        }
    }
}
```

当测试选择默认视图的操作方法时，会发生变化，如下所示：

```
...
public ViewResult Result() => View();
...
```

在这种情况下，需要确保视图名为null，如下所示：

```
...
Assert.Null(result.ViewName);
...
```

null值是ViewResult对象向MVC发出与操作方法关联的默认视图已被选择的方式。

通过路径指定视图

视图的命名约定方法虽然方便简单，但它限制了可以呈现的视图。如果要呈现特定的视图，可以通过提供明确的路径并绕过搜索阶段来实现。以下是一个通过路径指定视图的例子。

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            return View("/Views/Admin/Index");
        }
    }
}
```

当指定这样的视图时，路径必须以/或～/开头，并且可以包含文件扩展名（如果未指定，默认为.cshtml）。

如果发现自己使用了这个功能，建议花一点时间问问自己想要实现的目标。如果尝试呈现属于另一个控制器的视图，那么最好将用户重定向到该控制器中的操作方法（请参阅17.5.4节中的示例）。如果正在努力解决视图文件命名方案，那么请参阅第21章。

2．将数据从操作方法传递给视图

当使用ViewResult选择视图时，在生成HTML内容时，你可以通过操作方法向其传递要使用的数据。MVC为操作方法提供了将数据传递给视图的不同方式，这将在后面介绍。这些功能涉及第21章中的部分内容。本节仅讨论足够演示控制器功能的视图功能。

使用视图模型对象

通过将对象作为参数传递给View方法，你可以将创建的ViewResult对象传递给ViewData.Model属性。之前的章节直接设置了这个属性来解释POCO控制器是如何工作的，使用View方法会更简洁。代码清单17-18显示了新定义的 ExampleController类，将它添加到 Controllers 文件夹中，并将视图模型对象传递给 View方法。

代码清单17-18　Controllers文件夹下的ExampleController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() => View(DateTime.Now)
;
    }
}
```

以上代码将一个DateTime对象传递给View方法以用作视图模型。要从视图内访问对象，可使用Razor的Model关键字。创建Views/Example文件夹，并添加一个名为Index.cshtml的视图文件，内容如代码清单17-19所示。

代码清单17-19　Views/Example文件夹下的Index.cshtml文件的内容

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    Model: @(((DateTime)Model).DayOfWeek)
</body>
</html>
```

这是一个非类型化或弱类型的视图。视图不了

解视图模型对象的任何内容，并将其视为Object实例。为了获取DayOfWeek属性的值，需要将对象强制转换为DateTime实例，如下所示：

```
...
Model: @(((DateTime)Model).DayOfWeek)
...
```

虽然这样页面可以正常工作，但这可能产生混乱。可以通过创建强类型视图来整理这一切，其中的视图包括视图模型对象的类型细节，如代码清单17-20所示。

代码清单17-20　在Views/Example文件夹下的Index.cshtml文件中添加强类型视图

```
@model DateTime
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    Model: @Model.DayOfWeek
</body>
</html>
```

这里使用Razor的model关键字指定了视图模型的类型。请注意，在指定类型时使用关键字model，读取时使用关键字Model。

强类型不仅有助于整理视图，而且Visual Studio支持强类型视图的智能感知功能，如图17-7所示。

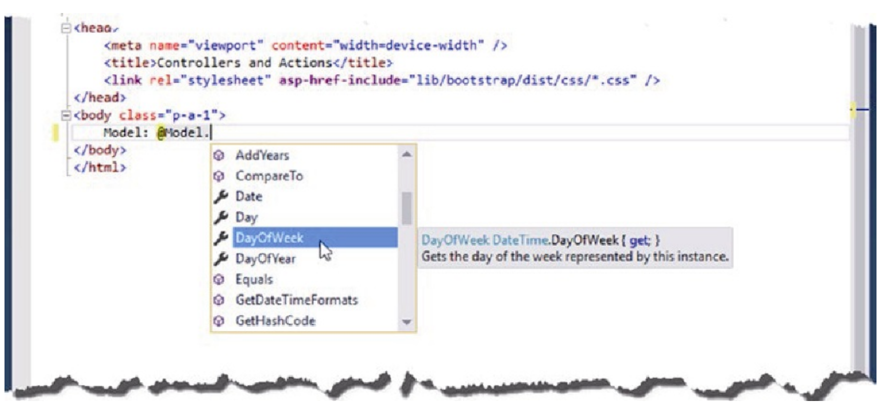


图17-7　强类型视图的智能感知功能

单元测试——查看模型对象

视图模型对象被分配给ViewResult.ViewData.Model属性，这意味着可以在使用View方法时测试操作方法是否发送预期的数据。下面的测试方法用于检查代码清单17-20中的操作方法的模型类型。

```
...
[Fact]
public void ModelObjectType() {
    //Arrange
    ExampleController controller = new ExampleControlle
r();

    // Act
    ViewResult result = controller.Index();

    // Assert
    Assert.IsType<System.DateTime>(result.ViewData.Mode
l);
}
...
```

Assert.IsType方法用于检查视图模型对象是否是DateTime实例。

在使用View方法时，有一个需要注意的问题：当希望使用与某个操作关联的默认视图并向该视图提供字符串模型对象时，就会出现这种情况，如代码清单17-21所示。

代码清单17-21　在Controllers文件夹下的ExampleController.cs文件中使用View方法

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() => View(DateTime.Now)
;

        public ViewResult Result() => View("Hello World
");
    }
```

```
}
```

在新的Result操作方法中，作者希望使用呈现操作的默认视图的View方法，并指定模型数据。但是，如果运行应用程序并请求URL /Example/Result，你将会看到类似下面的错误：

```
InvalidOperationException: The view 'Hello, World' was
not found.
The following locations were searched:
/Views/Example/Hello, World.cshtml
/Views/Shared/Hello, World.cshtml
```

问题是，由于字符串参数被解释为要呈现的视图的名称，因此MVC尝试查找名为“Hello,World.cshtml”的视图文件，而不是查找Result.cshtml。只需要通过将模型数据强制转换为对象，就可以很容易地解决这个问题，如代码清单17-22所示。

代码清单17-22　在Controllers文件夹下的ExampleController.cs文件中选择正确的View方法

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() => View(DateTime.Now)
;

        public ViewResult Result() => View((object)"Hel
lo World");
    }
}
```

将模型数据显式转换为对象可确保调用View方法的正确版本，并呈现Result.cshtml文件。

3．通过**View Bag**传递数据

第2章介绍了View Bag功能。此功能允许定义动态对象的属性，并在视图中访问它们。通过Controller类提供的ViewBag属性可访问动态对象，如代码清单17-23所示。

代码清单17-23　在Controllers文件夹下的ExampleController.cs

文件中使用View Bag功能

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public ViewResult Result() => View((object)"Hel
lo World");
    }
}
```

上面通过对它们进行赋值定义了名为Message和Date的ViewBag属性。在此之前，没有这样的属性存在，我们也没有做任何创建它们的准备。为了在视图中读取数据，需要获得与操作方法中设置的相同的属性，如代码清单17-24所示。

代码清单17-24　在Views/Example文件夹下的Index.cshtml文件

中从View Bag中读取数据

```
@model DateTime
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <p>The day is: @ViewBag.Date.DayOfWeek</p>
    <p>The message is: @ViewBag.Message</p>
</body>
</html>
```

View Bag相比使用视图模型对象有一个优点，就是很容易将多个对象发送到视图。如果MVC仅支持视图模型，那么需要创建具有字符串和DateTime成员的新类型，才能获得相同的效果。

警　告

Visual Studio无法为任何动态对象（包括View Bag）提供智能感知支持，并且在呈现视图之前不会提示错误。

单元测试——**View Bag**

ViewResult.ViewData属性会返回一个字典，其关键字是由操作方法定义的ViewBag属性的名称。以下是用于代码清单17-24中的操作方法的测试方法。

```
...
```

```
[Fact]
public void ModelObjectType() {
    //Arrange
    ExampleController controller = new ExampleControlle
r();

    // Act
    ViewResult result = controller.Index();

    // Assert
    Assert.IsType<string>(result.ViewData["Message"]);
    Assert.Equal("Hello", result.ViewData["Message"]);
    Assert.IsType<System.DateTime>(result.ViewData["Dat
e"]);
}
...
```

这个测试方法使用Assert.IsType方法检查Message和Date属性的类型，并使用Assert.Equal方法检查Message属性的值。

17.5.4　执行重定向

操作方法的常见结果是不直接产生任何输出，

而是将客户端重定向到另一个URL。大多数情况下，这个URL是应用程序中的另一个操作方法，用于生成希望用户查看的输出。执行重定向时，可以向浏览器发送以下两个HTTP编码之一。

- HTTP状态码302，表示临时重定向。这是最常用的重定向类型，当使用Post/Redirect/Get模式时，HTTP状态码302是将要发送的编码。
- HTTP状态码301，表示永久重定向。应该谨慎使用，因为它指示HTTP状态码的收件人不要再次请求原始URL，并使用包含在重定向代码旁边的新URL。如有疑问，请使用临时重定向——发送HTTP状态码302。

有几个不同的操作结果可用于执行重定向，如表17-9所示。

表**17-9**　用于执行重定向的操作结果

名称	控制器方法	描述

RedirectResult	Redirect RedirectPermanent	通过HTTP状态码301或302发送响应，将客户端重定向到新的URL
LocalRedirectResult	LocalRedirect LocalRedirectPermanent	将客户端重定向到本地URL
RedirectToActionResult	RedirectToAction RedirectionToActionPermanent	将客户端重定向到特定的操作和控制器
RedirectToRouteResult	RedirectToRoute RedirectToRoutePermanent	将客户端重定向到从特定路由生成的URL

1. 重定向到文本URL

重定向浏览器的最基本方法是调用由控制器类提供的重定向方法，返回RedirectResult类的一个实例，如代码清单17-25所示。

代码清单17-25　在Controllers文件夹下的ExampleController.cs文件中重定向到文本URL

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {
```

```
    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public ViewResult Result() => View((object)"Hel
lo World");

        public RedirectResult Redirect() => Redirect("/
Example/Index");
    }
}
```

重定向URL表示为重定向方法的字符串参数，以产生临时重定向。可以使用RedirectPermanent方法执行永久重定向，如代码清单17-26所示。

提　示

LocalRedirectionResult是可选的操作结果。如果控制器尝试重定向到任何非本地的URL，则会引发异常。当重定向到用户提供的URL时，这是很有用的。在这种情况下，重定向攻击会将另一个用户重定向到不受信任的站点。此类操作结果可以通过继承Controller类的LocalRedirect方法来创建。

代码清单17-26　在Controllers文件夹下的ExampleController.cs文件中永久重定到URL

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public ViewResult Result() => View((object)"Hel
```

```
lo World");

        public RedirectResult Redirect() => RedirectPer
manent("/Example/Index");
    }
}
```

单元测试——文本重定向

文本重定向很容易测试。可以读取URL，并使用RedirectResult类的Url和Permanent属性来测试重定向是永久的还是临时的。下面是用于代码清单17-26所示的永久重定向的测试方法：

```
...
[Fact]
public void Redirection() {
    // Arrange
    ExampleController controller = new ExampleControlle
r();
    // Act
    RedirectResult result = controller.Redirect();
    // Assert
    Assert.Equal("/Example/Index", result.Url);
    Assert.True(result.Permanent);
}
```

```
...
```

请注意，当调用操作方法时，已经更新了接收RedirectResult的测试。

2．重定向到路由系统URL

如果要将用户重定向到应用程序的其他部分，那么需要确保发送的URL在URL模式中有效。使用文本URL进行重定向的问题是，路由模式中的任何更改都意味着需要查看代码并更新URL。幸运的是，可以使用RedirectToRoute方法根据路由系统生成有效的URL，该方法创建了RedirectToRouteResult实例，如代码清单17-27所示。

提　示

如果正在按照本章中的示例顺序执行，那么可能必须清除浏览器的历史记录，代码清单17-27中的代码才能正常工作。这是因为浏览器会记住代码清单17-26中的永久重定向，并将对/example/Redirect的请求转换为对/Example/Index的请求，而不必再联系服务器。

代码清单17-27　在Controllers文件夹下的ExampleController.cs文件中重定向到路由系统URL

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {
```

```
        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public ViewResult Result() => View((object)"Hel
lo World");

        public RedirectToRouteResult Redirect() =>
            RedirectToRoute(new { controller = "Example
",
                                  action = "Index",
                                  ID = "MyID" });
    }
}
```

你可使用RedirectToRoute方法发出临时重定向，使用RedirectToRoutePermanent方法进行永久重定向。这两个方法都采用匿名类型，属性然后被传递给路由系统以生成URL，如第16章所述。

单元测试——路由重定向

以下是用于代码清单17-27中的操作方法的单

元测试:

```
...
[Fact]
public void Redirection() {
    // Arrange
    ExampleController controller = new ExampleControlle
r();
    // Act
    RedirectToRouteResult result = controller.Redirect(
);
    // Assert
    Assert.False(result.Permanent);
    Assert.Equal("Example", result.RouteValues["control
ler"]);
    Assert.Equal("Index", result.RouteValues["action"]);

    Assert.Equal("MyID", result.RouteValues["ID"]);
}
...
```

你可通过查看RedirectToRouteResult对象提供的路由信息来间接测试结果，这意味着不必解析URL，这将需要单元测试来对应用程序使用的URL模式进行假设。

3．重定向到操作方法

可以通过使用RedirectTo操作方法（用于临时重定向）或RedirectToActionPermanent方法（用于永久重定向），更加优雅地重定向到操作方法。这些都是对RedirectToRoute方法的包装，可以让你为操作方法和控制器指定值，而无须创建匿名类型，如代码清单17-28所示。

代码清单17-28　在Controllers文件夹下的ExampleController.cs文件中使用RedirectToAction方法进行重定向

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public RedirectToActionResult Redirect() => Red
```

```
irectToAction(nameof(Index));
    }
}
```

如果只指定一个操作方法，那么默认是当前控制器中的操作方法。如果要重定向到另一个控制器，则需要提供控制器的名称作为参数，如下所示：

```
...
public RedirectToActionResult Redirect()
    => RedirectToAction(nameof(HomeController), nameof(
HomeController.Index));
...
```

还有其他重载版本的方法，可以用来为URL提供其他的值。这些方法使用匿名类型，虽然这样会违背使用这些方法的目的，但它可以使代码更容易阅读。

注　意

为操作方法和控制器提供的值在传递到路由系统之前不会被验证。你有责任确保指定的目标实际存在。

单元测试——操作方法重定向

以下是用于代码清单17-28中的操作方法的单元测试：

```
...
[Fact]
public void Redirection() {
    // Arrange
    ExampleController controller = new ExampleControlle
```

```
r();
    // Act
    RedirectToActionResult result = controller.Redirect
();
    // Assert
    Assert.False(result.Permanent);
    Assert.Equal("Index", result.ActionName);
}
...
```

RedirectToActionResult类提供了ControllerName和ActionName属性，可以轻松地在控制器中创建重定向，而无须解析URL。

4．使用**Post/Redirect/Get**模式

重定向经常用于处理HTTP POST请求的操作方法。正如之前解释的那样，当需要更改应用程序的状态时，会使用POST请求。如果在处理POST请求后返回HTML响应，那么用户将有可能单击浏览器中的重新加载按钮并再次提交表单，这可能会产生

不可预料的结果。

可以在示例应用程序的Home控制器中看到此问题。ReceiveForm方法会从表单数据的参数中接收值，并使用View方法返回一个ViewResult对象：

```
...
public ViewResult ReceiveForm(string name, string city)
    => View("Result", $"{name} lives in {city}");
...
```

要查看问题，请运行应用程序并请求URL——/Home。提交表单，然后单击浏览器中的重新加载按钮。可使用F12键来研究浏览器发出的HTTP请求，你将看到一个新的POST请求被发送到服务器。这在简单的应用程序中没有什么影响，但如果POST请求最终重复删除数据，提交订单或执行用户未打算的其他重要任务，那么这个问题可能会造成严重后果。

为了避免这个问题，可以按照Post/Redirect/Get模式进行操作。在这种模式中，你将收到POST请求，处理它，然后重定向浏览器，以便浏览器为另一个URL发出GET请求。GET请求不应该修改应用程序的状态，因此在无意中重新提交请求不会导致任何问题。代码清单17-29添加了重定向，以便浏览器被重定向到具有GET请求的不同URL。

代码清单17-29　在Controllers文件夹下的HomeController.cs文件中使用Post/Redirect/Get模式

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("SimpleForm")
;

        [HttpPost]
        public RedirectToActionResult ReceiveForm(strin
g name, string city)
            => RedirectToAction(nameof(Data));
```

```
        public ViewResult Data() => View("Result");
    }
}
```

RedirectToActionResult方法通过POST请求从用户接收数据，并将客户端重定向到Data操作方法。如果用户重新加载页面，无害的GET请求将被发送到Data操作方法。第20章将介绍使用HttpPost特性可确保只有POST请求可以发送到ReceiveForm操作方法。

5．使用temp数据

重定向将导致浏览器发送全新的HTTP请求，这意味着无法从原始请求访问表单数据，还意味着Data方法不知道应该向用户显示的name和city值。

如果需要将数据从一个请求保存到另一个请求，则可以使用temp数据功能。temp数据与第9章

使用的会话数据类似，只是在处理请求时，temp数据值在读取和删除数据存储区时被标记为删除。对于在Post/Redirect/Get模式中进行重定向工作所需的短期数据，这是理想的安排。temp数据功能可通过名为TempData的控制器类属性获得，如代码清单17-30所示。

注　意

temp数据依赖于会话中间件。有关Startup类中对应功能的配置语句，可参见本章的开始部分。

代码清单17-30　在Controllers文件夹下的HomeController.cs文

件中使用temp数据

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            return View("SimpleForm");
        }

        [HttpPost]
        public RedirectToActionResult ReceiveForm(strin
g name, string city) {
            TempData["name"] = name;
            TempData["city"] = city;
            return RedirectToAction(nameof(Data));
        }

        public ViewResult Data() {
            string name = TempData["name"] as string;
            string city = TempData["city"] as string;
            return View("Result", $"{name} lives in {ci
ty}");
        }
    }
}
```

ReceiveForm方法使用TempData属性获取数据

字典，以便在将客户端重定向到Data操作方法之前存储name和city值。Data操作方法使用相同的TempData属性检索数据值，并使用它们创建将由视图显示的模型数据。

提　示

TempData字典还提供了Peek方法，以允许获取数据值而不将它们标记为删除。TempData字典还提供了Keep方法，可用于防止删除以前的读取值。Keep方法不会永远保护值。如果再次读取该值，就会再次将其标记为删除。请使用会话数据存储值，这样在处理请求时这些值才不会被删除。

17.5.5 返回不同类型的内容

操作方法并不一定必须响应为HTML，表17-10显示了可用于不同类型数据的内置操作结果。

表**17-10** 操作结果

操作结果	控制器方法	描述
JsonResult	Json	将对象序列化为JSON并将其返回给客户端
ContentResult	Content	发送包含指定对象的响应
ObjectResult	Not Available	将使用内容协商将对象发送给客户端
OkObjectResult	Ok	如果内容协商成功，就使用内容协商将HTTP对象发送给客户端
NotFoundObjectResult	NotFound	如果内容协商成功，就使用内容协商将对象发送到具有HTTP 404状态码的客户端

1．生成**JSON**响应

JSON（JavaScript对象表示法）格式已成为在Web应用程序及其客户端之间传输数据的标准方法。作为一种数据交换格式，JSON在很大程度上取代了XML，因为JSON更易于使用，特别是在编写客户端JavaScript时，因为JSON与JavaScript用于定义文本数据值的语法密切相关。第20章将介绍JSON在Web应用程序中的作用，代码清单17-31显示了如何使用Json方法创建JsonResult对象。

代码清单17-31　在Controllers文件夹下的ExampleController.cs文件中生成JSON响应

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public JsonResult Index() => Json(new[] { "Alic
e", "Bob", "Joe" });
    }
}
```

运行示例并请求URL——/Example，你将看到操作方法的响应，响应为JSON格式的C#字符串数组，如下所示：

```
["Alice","Bob","Joe"]
```

大多数浏览器可以直接显示JSON结果，但在其他浏览器（包括Microsoft Internet Explorer）中需要将数据保存到文件中才能查看。

单元测试——非HTML操作结果

重要的是要记住，操作方法的单元测试应该集中于返回格式化的数据而不是格式本身，这是由MVC处理的，通常超出大多数测试项目的范围。例如，以下是用于代码清单17-32中的操作方法的单元测试：

```
...
[Fact]
public void JsonActionMethod() {
    // Arrange
    ExampleController controller = new ExampleControlle
r();
    // Act
    JsonResult result = controller.Index();
    // Assert
    Assert.Equal(new[] { "Alice", "Bob", "Joe" }, resul
t.Value);
}
...
```

JsonResult类提供了Value属性，以返回将被转换成JSON的数据，产生返回客户端的响应。在单元测试中，可对Value属性与预期的数据进行比较。

2．使用对象生成响应

许多应用程序只需要来自控制器的HTML和JSON响应，其他类型的内容则依赖于静态文件来

传递，比如图像、JavaScript文件和CSS样式表。但是，当需要在响应中返回特定的内容类型时，就需要使用操作结果来帮助解决这些问题。最简单的是通过Content方法创建的ContentResult类，该类用于发送带有可选MIME内容类型的字符串值。在代码清单17-32中，使用Content方法重新创建了前面的JSON结果。

代码清单17-32　在Controllers文件夹下的ExampleController.cs文件中手动创建JSON结果

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ContentResult Index()
            => Content("[\"Alice\",\"Bob\",\"Joe\"]", "
application/json");
    }
}
```

当内容方便使用字符串格式，并且知道客户端

能够接收指定的MIME类型时，这种类型的操作结果非常有用。这种方法的问题在于不知道如何使用未知格式的数据向客户端发送响应。更健壮的方法依赖于内容协商，这是由ObjectResult执行的，如代码清单17-33所示。

代码清单17-33　在Controllers文件夹下的ExampleController.cs文件中使用内容协商

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ObjectResult Index() => Ok(new string[]
{ "Alice", "Bob", "Joe" });
    }
}
```

内容协商构建了一种复杂的系统，用于在浏览器和应用程序之间找出共同的格式。当浏览器发出HTTP请求时，已包括接收标头，用以指示可以处

理的格式。下面是用来测试示例的Google Chrome版本的标头：

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

支持的格式已表示为MIME类型。MVC有一组可用于数据值的格式，可与浏览器支持的格式做比较。MVC使用的首选格式是JSON，大部分时间将使用这种格式，除非使用明文返回字符串值。有关内容协商的过程及实现方式的更多详细信息，请参见第20章。

17.5.6　响应文件的内容

大多数应用程序依赖于静态文件中间件传递文件的内容，但还有一组可用于将文件发送到客户端的操作结果，如表17-11所示。

警　告

使用这些操作结果时要小心，并确保不创建允许请求任意文件内容的应用程序。特别是，不要从请求的任何部分或用户可以通过请求修改的任何数据存储中获取要发送的文件的路径。

表**17-11**　将文件发送到客户端的操作结果

操作结果	控制器方法	描述
FileContentResult	File	使用指定的MIME类型向客户端发送字节数组
FileStreamResult	File	读取流并将内容发送给客户端
VirtualFileResult	File	从虚拟路径读取流（相对于主机上的应用程序）
PhysicalFileResult	PhysicalFile	从指定的路径读取文件的内容，并将内容发送给客户端

代码清单17-34使用继承自Controller类的File方法返回Bootstrap CSS文件，并作为Example控制器上的Index操作方法的返回结果。

代码清单17-34　在Controllers文件夹下的ExampleController.cs文件中使用文件作为响应

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public VirtualFileResult Index()
            => File("/lib/bootstrap/dist/css/bootstrap.
css", "text/css");
    }
}
```

为了使用这个操作方法，这里已经使用Url辅助方法修改了SimpleForm.cshtml文件中的link元素，如代码清单17-35所示。

代码清单17-35　在SimplerForm.cshtml文件中指向一个操作方法

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Controllers and Actions</title>
    <link rel="stylesheet" href="@Url.Action("Index", "
Example")" />
</head>
<body class="m-1 p-1">
    <form method="post" asp-action="ReceiveForm">
        <div class="form-group">
            <label for="name">Name:</label>
            <input class="form-control" name="name" />
        </div>
        <div class="form-group">
            <label for="name">City:</label>
            <input class="form-control" name="city" />
        </div>
        <button class="btn btn-primary center-block" ty
pe="submit">Submit</button>
    </form>
</body>
</html>
```

如果运行示例并请求URL——/Home，则发送到浏览器的HTML响应将包含以下元素：

```
<link rel="stylesheet" href="/Example" />
```

这将会使浏览器发送针对代码清单17-35中的操作方法的HTTP请求，进而发送在视图中为内容设置样式所需的CSS文件。

注　意

标签助手是提供CSS的非常有用的工具，将在第25章中介绍。

17.5.7　返回错误和HTTP状态码

内置的ActionResult类的最终集合（其成员见表17-12）可用于向客户端发送特定的错误消息和

HTTP状态码。大多数应用程序不需要这些功能，因为ASP.NET Core和MVC将自动生成这些类型的结果。但是，如果需要对发送给客户端的响应进行更直接的控制，它们将非常有用。

表**17-12**　内置的**ActionResult**类的最终集合的成员

成员	控制器方法	描述
StatusCodeResult	StatusCode	向客户端发送指定的HTTP状态码
OkResult	Ok	向客户端发送HTTP 200状态码
CreatedResult	Created	向客户端发送HTTP 201状态码
CreatedAtActionResult	CreatedAtAction	将HTTP 201状态码发送到客户端，并在Location标头中指向action和控制器的URL
CreatedAtRouteResult	CreatedAtRoute	将HTTP 201状态码发送到客户端，并将URL放在从特定路由生成的Location标头中
BadRequestResult	BadRequest	向客户端发送HTTP 400状态码
UnauthorizedResult	Unauthorized	向客户端发送HTTP 401状态码

NotFoundResult	NotFound	向客户端发送HTTP 404状态码
UnsupportedMediaTypeResult	None	向客户端发送HTTP 415状态码

1. 发送特定的HTTP状态码

若使用StatusCode方法向浏览器发送特定的HTTP状态码，将创建一个StatusCodeResult对象，如代码清单17-36所示。

代码清单17-36 在Controllers文件夹下的ExampleController.cs文件中发送特定的状态码

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public StatusCodeResult Index()
            => StatusCode(StatusCodes.Status404NotFound
);
    }
}
```

StatusCode方法接收一个int值，可以使用它直接指定状态码。Microsoft.AspNetCore.Http命名空间中的StatusCodes类定义了HTTP支持的所有状态码的字段。在以上代码中，已使用Status404NotFound字段返回HTTP状态码404，这表示请求的资源不存在。

2．发送**404**结果

表17-12中显示的其他操作结果扩展或依赖StatusCodeResult类，该类提供了一种更方便的发送特定状态码的方法。可以使用更方便的NotFoundResult类来实现与代码清单17-36相同的效果，该类是从StatusCodeResult类派生的，可以使用控制器的NotFound方法创建，如代码清单17-37所示。

代码清单17-37　在Controllers文件夹下的ExampleController.cs

文件中生成404结果

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public StatusCodeResult Index() => NotFound();
    }
}
```

测试HTTP状态码

StatusCodeResult类遵循已为其他结果类型使用的模式，并通过一组属性使其状态可用。在这种情况下，StatusCode属性返回HTTP状态码，StatusDescription属性返回关联的描述性字符串。下面的测试方法可用于代码清单17-37中的操作方法。

```
...
[Fact]
public void NotFoundActionMethod() {
    // Arrange
    ExampleController controller = new ExampleControlle
r();
    // Act
    StatusCodeResult result = controller.Index();
    // Assert
    Assert.Equal(404, result.StatusCode);
}
...
```

17.5.8　理解其他操作结果类

一些其他的操作结果类与其他章节中描述的MVC功能紧密相连，表17-13列出了这些类。

表17-13　其他操作结果类

名称	控制器方法	描述
PartialViewResult	PartialView	用于选择分部视图

ViewComponentResult	ViewComponent	用于选择视图组件
EmptyResult	None	不执行任何工作，并生成返回客户端的空响应
ChallengeResult	None	用于在请求中强制实施安全策略

17.6　小结

控制器是MVC设计模式中的关键构件之一，是MVC开发的核心。在本章，你已经了解了如何使用基本的C#类创建POCO控制器，以及如何利用Controller基类进行快速开发。你看到了操作结果在MVC控制器中扮演的角色，以及它们如何简化单元测试。本章还展示了可以从操作方法接收输入和生成输出的不同方式，并演示了内置的操作结果。下一章将描述导致ASP.NET开发人员困惑的特性之一，即依赖注入，但它对于有效的MVC开发是必不可少的。

第18章　依赖注入

本章将介绍依赖注入（Dependency Injection，DI），这是一种有助于灵活创建应用和简化单元测试的技术。不论从依赖注入有什么作用看，还是从依赖注入如何执行看，依赖注入都是一个很难理解的主题。因此，本章从传统的构建应用程序组件的方式开始，逐步解释依赖注入的工作原理及其重要性。表18-1展示了在上下文中使用的依赖注入。

表**18-1**　在上下文中使用的依赖注入

问题	答案
什么是依赖注入？	依赖注入可以轻松创建松散耦合的组件，这通常意味着组件使用接口定义的功能，而不需要关注功能具体由哪些类来实现
依赖注入有什么作用？	依赖注入可以通过更改实现了定义应用程序功能的接口的组件来更容易地更改应用程序的行为，还会导致更易于隔离单元测试的组件

如何使用依赖注入？	Startup类用于指定哪些实现类用于传递由应用程序使用的接口指定的功能。当对象（如控制器）创建新的处理请求时，将自动提供它们所需的实现类的实例
有什么陷阱或限制吗？	主要的限制是，类将服务的使用声明为构造函数参数，这可能导致构造函数的唯一作用是接收依赖项并将它们分配给实例字段
有其他的选择吗？	不必在自己的代码中使用依赖注入，但是了解依赖注入的工作原理是很有帮助的，因为MVC使用依赖注入为开发人员提供功能

表18-2展示了本章要介绍的操作。

表**18-2**　本章要介绍的操作

操作	方法	代码清单
创建松散耦合的组件	通过接口隔离类，并使用外部映射将它们连接在一起	代码清单18-9～代码清单18-16
在组件（比如控制器）中声明依赖项	定义组件需要的类型的构造函数参数	代码清单18-17
配置服务映射	在Startup类中添加映射	代码清单18-18，代码清单18-20～代码清单18-26
对具有依赖项的组	创建服务接口的模拟实现，并在单元测试	

件进行单元测试	中创建组件时作为构造函数参数传递	代码清单18-19
指定实现对象的创建方式	使用适合要管理的服务的生命周期方法创建服务映射	代码清单18-27～代码清单18-31
在控制器中接收单个操作方法的依赖项	使用操作注入	代码清单18-32
在控制器中手动请求实现对象	使用HttpContext.RequestServices属性	代码清单18-33

18.1　准备示例项目

在本章中，使用ASP.NET Core Web Application（.NET Core）模板创建一个新的名为DependencyInjection的Empty项目。代码清单18-1显示了Startup类，用于配置项目的服务和中间件。

代码清单18-1　DependencyInjection文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace DependencyInjection {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

18.1.1　创建模型和存储库

创建Models文件夹并添加一个名为Product.cs的类文件，内容如代码清单18-2所示。

代码清单18-2　Models文件夹下的Product.cs文件的内容

}

```
namespace DependencyInjection.Models {

    public class Product {

        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

为了管理模型，在Models文件夹中添加一个名为IRepository.cs的类文件，其中定义的接口如代码清单18-3所示。

代码清单18-3　Models文件夹下的IRepository.cs文件的内容

```
using System.Collections.Generic;

namespace DependencyInjection.Models {

    public interface IRepository {

        IEnumerable<Product> Products { get; }

        Product this[string name] { get; }

        void AddProduct(Product product);
        void DeleteProduct(Product product);
    }
}
```

IRepository接口定义了可对Product对象集合执行的操作。为了提供这个接口的实现，在Models文件夹中添加一个名为MemoryRepository.cs的类文件，内容如代码清单18-4所示。

代码清单18-4　Models文件夹下的MemoryRepository.cs文件的内容

```
using System.Collections.Generic;

namespace DependencyInjection.Models {

    public class MemoryRepository : IRepository {
        private Dictionary<string, Product> products;

        public MemoryRepository() {
            products = new Dictionary<string, Product>(
);
            new List<Product> {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M }
            }.ForEach(p => AddProduct(p));
        }

        public IEnumerable<Product> Products => product
```

```
s.Values;

        public Product this[string name] => products[na
me];

        public void AddProduct(Product product) =>
            products[product.Name] = product;

        public void DeleteProduct(Product product) =>
            products.Remove(product.Name);
    }
}
```

MemoryRepository类使用字典将模型对象存储在内存中。这意味着没有持久存储，在停止或重新启动应用程序时，会将模型重置为在构造函数中创建的示例数据对象。对于真正的项目来说，这不是合理的方法，但用在这里很合适，本章的重点在于介绍应用程序的各个不同方面是如何工作的。

18.1.2　创建控制器和视图

创建Controllers文件夹，添加一个名为HomeController.cs的类文件，内容如代码清单18-5

所示。

代码清单18-5　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View();
    }
}
```

Home控制器只有一个操作方法，它使用View方法创建一个用于渲染默认视图的ViewResult对象。要创建与操作方法关联的视图，可创建Views/Home文件夹，并在其中添加一个名为Index.cshtml的Razor视图文件，内容如代码清单18-6所示。

代码清单18-6　Views/Home文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Dependency Injection</title>
    <link rel="stylesheet" asp-href-include="lib/bootst
rap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    @if (ViewData.Count > 0) {
        <table class="table table-bordered table-sm tab
le-striped">
            @foreach (var kvp in ViewData) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td
></tr>
            }
        </table>
    }
    <table class="table table-bordered table-sm table-s
triped">
        <thead>
            <tr><th>Name</th><th>Price</th></tr>
        </thead>
        <tbody>
            @if (Model == null) {
                <tr><td colspan="3" class="text-center"
>No Model Data</td></tr>
            } else {
                @foreach (var p in Model) {
                    <tr>
                        <td>@p.Name</td>
```

```
                        <td>@string.Format("{0:C2}", p.
Price)</td>
                    </tr>
                }
            }
        </tbody>
    </table>
</body>
</html>
```

Index视图使用Product对象的枚举进行强类型化，视图的主要内容是HTML表格。如果控制器没有提供任何类型的数据，则会显示一条消息作为表格的唯一内容。如果存在模型数据，则会为枚举中的每个Product对象向表中添加一行。还有一个表格，用于在视图中枚举View Bag的键和值（如果View Bag的键和值不存在，则隐藏这个表格）。

Index视图依赖于Bootstrap CSS包来为HTML元素设置样式。要将Bootstrap添加到项目中，可使用Bower Configuration File模板创建bower.json文件，并将Bootstrap软件包添加到bower.json的

dependencies部分，如代码清单18-7所示。

代码清单18-7　在DependencyInjection文件夹下的bower.json文件中添加Bootstrap

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

最后的准备工作是在Views文件夹中创建_ViewImports.cshtml文件，该文件设置了用于Razor视图的内置标签助手，并导入模型命名空间，如代码清单18-8所示。

代码清单18-8　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using DependencyInjection.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

18.1.3　创建单元测试项目

按照第7章中描述的步骤，使用xUnit Test Project（.NET Core）模板创建一个名为DependencyInjection.Tests的项目。这里删除了unittest1. cs文件，因此项目中目前没有任何测试。如果运行应用程序，你将看到图18-1所示的结果。

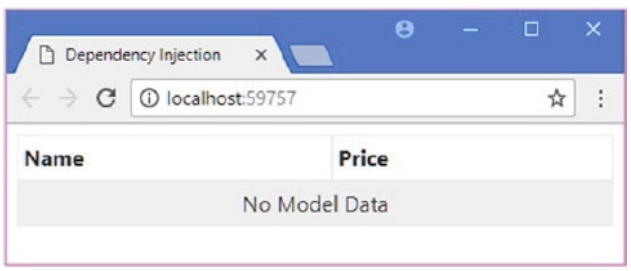


图18-1　运行示例应用程序的结果

18.2　创建松散耦合的组件

图18-1没有显示模型数据的原因在于需要将模型数据传递给视图的HomeController类与包含模型数据的MemoryRepository类之间没有关系。在MVC

应用程序中，将组件连接在一起的目标是希望能够使用相同功能的替代实现轻松替换组件。

通过更换组件可以进行有效的单元测试，轻松地在不同的托管环境（如开发和生产服务器）中更改应用程序的行为，并简化长期的应用程序维护。

接下来的几节将首先解释替代方法及其带来的问题。这似乎是解释依赖注入（DI）特性的一种间接方式，但是DI面临的挑战在于解决一些在编写代码时不明显，但在开发周期的末段才出现的问题。

了解依赖注入

依赖注入可能是一个难以理解的话题，DI是一个有用的工具，但不是每个人都喜欢或需要。

如果没有进行单元测试，或者正在开展小型、独立且稳定的项目，DI将只能提供有限的好处。了解DI的工作原理仍然有帮助，因为DI用于访问某些重要的MVC功能，但你并不总是需要在控制器和其他类中使用DI。

推荐你在自己的项目中使用DI，主要是因为作者发现项目经常出现一些意想不到的问题，如果能够轻松地用新的实现替换组件，就可以避免大量烦琐且容易出错的更改。

检查紧密耦合的组件

对于大多数开发人员来说，自然倾向是采取最直接的途径来解决问题。对于示例应用程序来说，这意味着使用new关键字创建控制器所需的存储库对象，以获取模型数据，如代码清单18-9所示。

代码清单18-9 在Controllers文件夹下的HomeController.cs文件中实例化存储库

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View(new MemoryRep
ository().Products);
    }
}
```

好消息是，这段代码是有效的。如果运行应用程序，你将看到浏览器中显示的模型对象的详细信息，如图18-2所示。

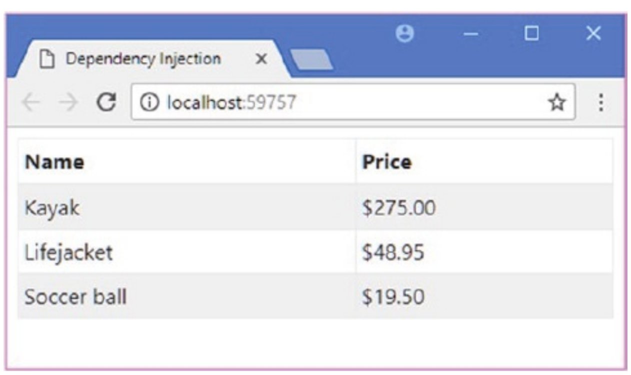


图18-2　模型对象的详细信息

但坏消息是，Home控制器和MemoryRepository类现在紧密耦合，这意味着无法在不改变HomeController类的前提下直接替换存储库。正如第7章所解释的，执行有效的单元测试意味着能够隔离单个组件，但是在测试代码清单18-9中的Index操作方法时会隐式地测试存储库类。如果单元测试失败，将无法确认问题出在控制器、存储库还是存储库依赖的其他组件上。所以根据当前实际情况，Home控制器和MemoryRepository形成了单独的单元，紧耦合组件的影响如图18-3所示。

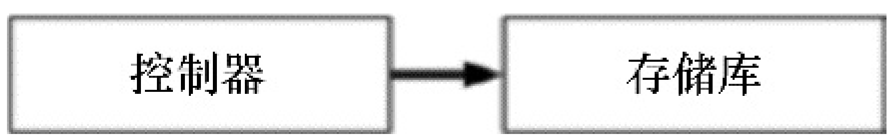


图18-3　紧耦合组件的影响

1．为单元测试解耦合

第7章使用一个属性，通过实现的接口来存储

对存储库类的引用，从而允许为单元测试创建模拟库。代码清单18-10显示了在本章的示例应用程序中，如何在控制器中使用这种方法。

代码清单18-10　在Controllers文件夹下的HomeController.cs文件中使用Repository属性

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {

        public IRepository Repository { get; set; } = n
ew MemoryRepository();

        public ViewResult Index() => View(Repository.Pr
oducts);
    }
}
```

如果要进行单元测试，那么这种方法非常适用，因为可以通过在单元测试中调用操作方法之前设置Repository属性来隔离控制器类。

将一个名为DITests.cs的类文件添加到DependencyInjection.Tests项目中，并用来定义代码清单18-11所示的单元测试，从而使用Repository属性在对控制器执行操作之前设置假的存储库。

代码清单18-11　在单元测试项目中使用DITests.cs文件测试控制器

```
using DependencyInjection.Controllers;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {

    public class DITests {

        [Fact]
        public void ControllerTest() {
            // Arrange
            var data = new[] { new Product { Name = "Te
st", Price = 100 } };
            var mock = new Mock<IRepository>();
            mock.SetupGet(m => m.Products).Returns(data
);
            HomeController controller = new HomeControl
ler {
                Repository = mock.Object
```

```
            };

            // Act
            ViewResult result = controller.Index();

            // Assert
            Assert.Equal(data, result.ViewData.Model);
        }
    }
}
```

Repository属性允许隔离控制器并提供可以在操作方法创建的ViewResult中检查的测试数据。这仅提供紧密耦合的组件问题的部分解决方案，因为在应用程序运行时无法设置Repository属性。正如第17章所解释的那样，MVC负责实例化控制器来处理请求，它并不知道Repository属性的特殊重要性。该技术产生的效果是在执行单元测试时控制器和存储库松散耦合，但在应用程序运行时控制器和存储库紧密耦合，如图18-4所示。

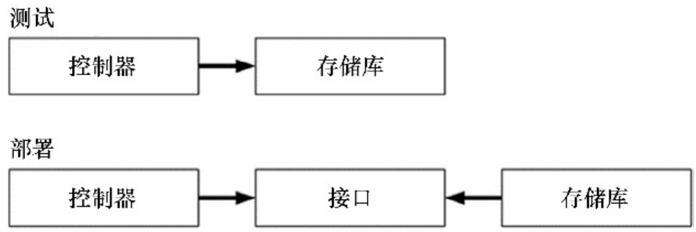


图18-4　添加Repository属性后的效果

2．使用类型代理

下一个合乎逻辑的步骤是决定将哪种存储库接口的实现用于控制器类，并放在应用程序的其他位置。为了演示这是如何工作的，在示例应用程序中添加Infrastructure文件夹，并在其中添加一个名为TypeBroker.cs的类文件，内容如代码清单18-12所示。

代码清单18-12　Infrastructure文件夹下的TypeBroker.cs文件的内容

```
using DependencyInjection.Models;
using System;

namespace DependencyInjection.Infrastructure {
    public static class TypeBroker {
```

```
        private static Type repoType = typeof(MemoryRep
ository);
        private static IRepository testRepo;

        public static IRepository Repository =>
            testRepo ?? Activator.CreateInstance(repoTy
pe) as IRepository;

        public static void SetRepositoryType<T>() where
 T : IRepository =>
            repoType = typeof(T);

        public static void SetTestObject(IRepository re
po) {
            testRepo = repo;
    }
}
```

TypeBroker类定义了Repository属性，并返回实现了IRepository接口的新对象。Repository属性使用的实现类由repoType字段的值确定，该字段默认为MemoryRepository，但可以通过调用SetRepositoryType方法进行更改。

为了支持单元测试，SetTestObject方法允许使用特定的对象。代码清单18-13更新了Home控制

器，以便从代理获取存储库对象。

代码清单18-13　在Controllers文件夹下的HomeController.cs文件中使用类型代理

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {

        public IRepository Repository { get; } = TypeBr
oker.Repository;

        public ViewResult Index() => View(Repository.Pr
oducts);
    }
}
```

示例应用程序中现在有一组更复杂的关系，如图18-5所示。需要注意的一点是，控制器类和存储库类之间没有直接的关系：一切都是通过接口和代理来调用的。这意味着可以更改存储库类，而不必对控制器进行任何更改。

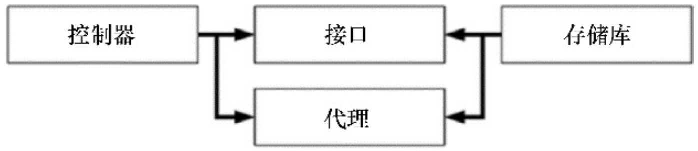


图18-5　添加类型代理后复杂的关系

为了演示类型代理的使用，在Models文件夹中添加一个名为AlternateRepository.cs的类文件，并用来定义IRepository接口的另一个实现，如代码清单18-14所示。

代码清单18-14　Models文件夹下的AlternateRepository.cs文件的内容

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
    public class AlternateRepository : IRepository {
        private Dictionary<string, Product> products;

        public AlternateRepository() {
            products = new Dictionary<string, Product>(
);
            new List<Product> {
                new Product { Name = "Corner Flags", Pr
ice = 34.95M },
                new Product { Name = "Stadium", Price =
 79500M }
```

```
            }.ForEach(p => AddProduct(p));
        }
        public IEnumerable<Product> Products => product
s.Values;

        public Product this[string name] => products[na
me];

        public void AddProduct(Product product) =>
            products[product.Name] = product;

        public void DeleteProduct(Product product) =>
            products.Remove(product.Name);
    }
}
```

在实际应用中，备用存储库可能以不同的格式存储数据，或者使用不同类型的持久化数据。在本例中，AlternateRepository类和MemoryRepository类之间的区别是在实例化类时创建的模型数据。要使用AlternateRepository类，可在Startup类的ConfigureServices方法中配置类型代理，如代码清单18-15所示。

代码清单18-15　在DependencyInjection文件夹下的Startup.cs文件中配置类型代理

会显示新的存储库类提供的数据，如图18-6所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            TypeBroker.SetRepositoryType<AlternateRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

可以通过启动应用程序来查看更改的效果，这会显示新的存储库类提供的数据，如图18-6所示。

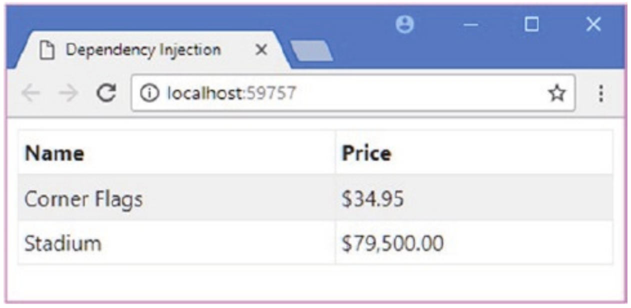


图18-6　更换存储库类

类型代理允许将特定对象用作存储库，这样就可以像代码清单18-16那样编写单元测试。

代码清单18-16　使用测试项目的DITests.cs文件中的代理进行测试

```
using DependencyInjection.Controllers;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {

    public class DITests {

        [Fact]
        public void ControllerTest() {
            // Arrange
            var data = new[] { new Product { Name = "Te
```

```
st", Price = 100 } };
            var mock = new Mock<IRepository>();
            mock.SetupGet(m => m.Products).Returns(data
);
            TypeBroker.SetTestObject(mock.Object);
            HomeController controller = new HomeControl
ler();

            // Act
            ViewResult result = controller.Index();

            // Assert
            Assert.Equal(data, result.ViewData.Model);
        }
    }
}
```

18.3 ASP.NET的依赖注入

18.2节介绍了分离控制器类和提供模型数据的存储库的过程。HomeController类现在可以获取IRepository接口的实现，而不知道正在使用哪个类或如何实例化。有关Irepository接口的信息包含在TypeBroker类中，可由任何其他需要访问存储库的控制器使用，也可用于其他测试对象。

虽然依赖注入可以构建更加灵活的应用，但也有一些不方便的地方。最大的缺点是开发人员必须为每个用来管理代理的新类型添加新的方法和属性。可以重写TypeBroker类来使其更通用，但是没有任何需要，因为ASP.NET Core提供相同功能的版本，并且更易于使用，不需要任何其他的类。

18.3.1　准备依赖注入

依赖注入描述了创建松散耦合组件的另一种方法，已被集成到ASP.NET Core平台中，由MVC自动使用，这意味着控制器和其他组件不需要知道如何创建它们所需的类型。代码清单18-17展示了应如何在Home控制器中使用DI。

代码清单18-17　准备在Controllers文件夹下的HomeController.cs文件中使用依赖注入

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
```

```
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) => repo
sitory = repo;

        public ViewResult Index() => View(repository.Pr
oducts);
    }
}
```

控制器将其依赖项声明为构造函数参数。这是依赖注入的第一部分，依赖注入中的依赖项是创建类的新实例所需的对象。在这种情况下，控制器类已声明对IRepository接口的依赖项。

在ASP.NET Core中，名为服务提供者的组件负责将接口映射到用于满足依赖关系的实现类型。

当需要新的控制器时，MVC会要求服务提供者创建一个新的HomeController实例。服务提供者

检查HomeController构造函数以确定其依赖性，创建所需的服务对象，并将它们注入HomeController构造函数以创建可用于处理请求的新控制器。这是依赖注入的核心过程，所以再详细介绍一遍。

（1）MVC接收针对Home控制器上的操作方法的传入请求。

（2）MVC向ASP.NET的服务提供者组件请求HomeController类的新实例。

（3）服务提供者检查HomeController构造函数，并发现它与IRepository接口有依赖关系。

（4）服务提供者查询其映射，以查找已被告知要用于依赖于IRepository接口的实现类。

（5）服务提供者创建实现类的新实例。

（6）服务提供者创建一个新的HomeController对象，使用实现对象作为构造函数参数。

（7）服务提供者将新创建的HomeController对象返回给MVC，MVC使用它处理传入的HTTP请求。

整体效果与前面的自定义类TypeBroker相同，但重要的优点是将依赖注入过程集成到了MVC中，这意味着在创建控制器类时将使用服务提供者组件。这允许控制器类声明依赖关系，而不需要知道如何解决它们。你只需要编写将其依赖性声明为构造函数参数的控制器类，并让MVC和服务提供者组件去完成其他的工作。

注　意

本章的所有示例都使用ASP.NET内置的依赖注入，它是ASP.NET Core的一部分。你可使用第三方软件包替换内置的依赖注入，这些包可以提供一些更强或更多的功能。热门的包括Autofac和StructureMap等，但在使用时需要将它们集成到ASP. NET Core中，你可以在GitHub网站上找到更多详细信息。

18.3.2　配置服务提供者

如果运行示例项目，你会发现通过HomeController构造函数声明依赖项已经破坏了应用程序。如果MVC尝试创建HomeController类的实例来为请求提供服务，就会遇到图18-7所示的错误。

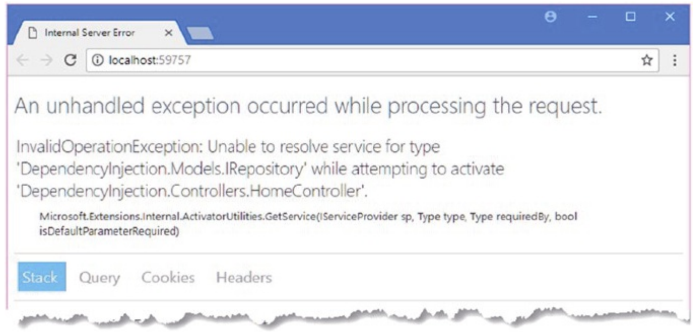


图18-7　运行示例项目出现的错误

要解决依赖项，必须配置服务提供者，以便它知道如何解决服务依赖关系。当前的服务提供者没有这些信息，当要求创建HomeController对象时，会抛出异常，因为不知道如何解决IRepository接口上的依赖关系。

服务提供者的配置是在Startup类中定义的，这样才能保证在应用程序开始接收请求之前服务已经准备就绪。代码清单18-18配置了服务提供者，从而使它知道如何处理IRepository接口上的依赖关系。

代码清单18-18　在DependencyInjection文件夹下的Startup.cs文

件中配置服务提供者

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IRepository, MemoryRe
pository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

依赖注入是使用你在ConfigureServices方法接收的IServiceCollection对象上调用的扩展方法进行配置的。以上代码中使用的AddTransient扩展方法会告诉服务提供者如何处理依赖关系（本章后面会详细描述）。映射使用类型参数表示，第一个类型是接口，第二个类型是实现类。

```
...
services.AddTransient<IRepository, MemoryRepository>();
...
```

上面的语句告诉服务提供者通过创建一个MemoryRepository对象来解析IRepository接口上的依赖关系。如果运行示例应用程序，你将看到由HomeController构造函数声明的依赖项被解析，并且控制器提供了对模型数据的访问，依赖注入的配置如图18-8所示。

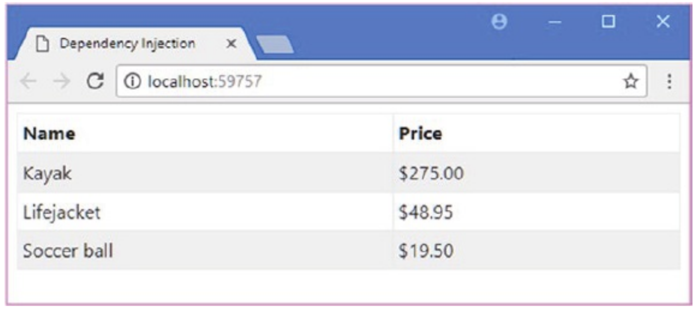


图18-8　依赖注入的配置

18.3.3　对具有依赖项的控制器进行单元测试

使用构造函数接收依赖关系可以使控制器的单元测试更容易。代码清单18-19显示了代码清单18-18中控制器的单元测试。

代码清单18-19　在单元测试项目中使用DITests.cs文件测试控制器

```
using DependencyInjection.Controllers;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {

    public class DITests {
```

```
        [Fact]
        public void ControllerTest() {
            // Arrange
            var data = new[] { new Product { Name = "Te
st", Price = 100 } };
            var mock = new Mock<IRepository>();
            mock.SetupGet(m => m.Products).Returns(data
);
            HomeController controller = new HomeControl
ler(mock.Object);

            // Act
            ViewResult result = controller.Index();

            // Assert
            Assert.Equal(data, result.ViewData.Model);
        }
    }
}
```

只要实现的接口正确，控制器就不需要知道或关心要将什么样的对象传递给构造函数。这允许使用测试用的存储库，而不必依赖任何可能影响测试结果的外部类（例如TypeBroker类）。

18.3.4 使用依赖关系链

当服务提供者需要解析依赖关系时，将检查已

配置用于查看是否具有解析依赖关系的类型。结果是可以创建依赖关系链，所有这些都是在运行时解析的，并且可以通过Startup类中的配置进行管理。为了演示依赖关系链，在Models文件夹中创建一个名为IModelStorage.cs的类文件，内容如代码清单18-20所示。

代码清单18-20　Models文件夹下的IModelStorage.cs文件的内容

```
using System.Collections.Generic;

namespace DependencyInjection.Models {

    public interface IModelStorage {
        IEnumerable<Product> Items { get; }
        Product this[string key] { get; set; }
        bool ContainsKey(string key);
        void RemoveItem(string key);
    }
}
```

IModelStorage接口定义了简单的对Product对象进行存储的行为。为了实现这个接口，在Models文

件夹中添加一个名为DictionaryStorage.cs的类文件，内容如代码清单18-21所示。

代码清单18-21　Models文件夹下的DictionaryStorage.cs文件的内容

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
    public class DictionaryStorage : IModelStorage {
        private Dictionary<string, Product> items
            = new Dictionary<string, Product>();

        public Product this[string key] {
            get { return items[key]; }
            set { items[key] = value; }
        }

        public IEnumerable<Product> Items => items.Valu
es;
        public bool ContainsKey(string key) => items.Co
ntainsKey(key);
        public void RemoveItem(string key) => items.Rem
ove(key);
    }
}
```

DictionaryStorage类通过使用强类型字典来存储模型对象，从而实现了IModelStorage接口。这是

当前包含在MemoryRepository类中的功能，在实际的项目中使用接口来分隔几乎没有什么价值，但它可以提供一个有用的示例，以说明如何使用依赖注入而不会使示例应用程序的复杂性大大增加。

代码清单18-22更新了MemoryRepository类，以便声明该类依赖IModelStorage接口，但它并不了解在实际运行时使用的实现类。

代码清单18-22　在Models文件夹下的MemoryRepository.cs文件中声明依赖关系

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
    public class MemoryRepository : IRepository {
        private IModelStorage storage;

        public MemoryRepository(IModelStorage modelStor
e) {
            storage = modelStore;
            new List<Product> {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
```

```
                new Product { Name = "Soccer ball", Pri
ce = 19.50M }
            }.ForEach(p => AddProduct(p));
        }

        public IEnumerable<Product> Products => storage
.Items;

        public Product this[string name] => storage[nam
e];

        public void AddProduct(Product product) =>
            storage[product.Name] = product;

        public void DeleteProduct(Product product) =>
            storage.RemoveItem(product.Name);
    }
}
```

如果运行示例应用程序，你将看到服务提供者会提示以下异常信息：

```
InvalidOperationException: Unable to resolve service fo
r type
'DependencyInjection.Models.IModelStorage' while attemp
ting to activate
'DependencyInjection.Models.MemoryRepository'.
```

这表明服务提供者正在使用依赖关系链。当要求创建一个新的控制器时，服务提供者检查

HomeController构造函数，并发现了具有依赖关系的IRepository接口，该接口知道应该使用一个MemoryRepository对象来解决问题。服务提供者随后检查MemoryRepository构造函数，该构造函数依赖于IModelStorage接口。但配置中没有指定如何解析IModelStorage依赖关系，这意味着无法创建MemoryRepository对象，反过来这意味着无法创建HomeController对象。所以服务提供者无法向MVC提供处理请求所需的对象，抛出异常。

因此，需要添加类型映射，告诉服务提供者如何解决依赖于IModelStorage的依赖关系，这里已经将类型映射添加到代码清单18-23所示的应用程序配置中。

代码清单18-23　在DependencyInjection文件夹下的Startup.cs文件中配置类型映射

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IRepository, MemoryRe
pository>();
            services.AddTransient<IModelStorage, Dictio
naryStorage>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

通过上面的代码，服务提供者可以满足依赖关

系链中的两个依赖关系，并且能够创建服务请求所需的一组对象——将注入MemoryRepository构造函数的DictionaryStorage对象注入HomeController构造函数。依赖关系链不仅很智能，它们还允许通过组合轻松隔离测试的组件来组合复杂的功能，以及轻松地更改以适应项目逐渐成熟的不断变化的需求。

18.3.5　对具体类型使用依赖注入

依赖注入也可以用于不能通过接口访问的具体类型。虽然这并没有提供使用接口的松散耦合优势，但它是十分有用的技术，因为允许在应用程序中的任何地方访问对象，并将具体的类型放在生命周期管理中，这将在后面具体介绍。

为了演示，在Models文件夹添加一个名为ProductTotalizer.cs的类文件，内容如代码清单18-24所示。

代码清单18-24　Models文件夹下的ProductTotalizer.cs文件的内容

```
using System.Linq;

namespace DependencyInjection.Models {
    public class ProductTotalizer {

        public ProductTotalizer(IRepository repo) => Re
pository = repo;

        public IRepository Repository { get; set; }

        public decimal Total => Repository.Products.Sum
(p => p.Price);
    }
}
```

ProductTotalizer类没有什么特别的用途，但它对IRepository接口具有依赖性，这意味着通过使用依赖注入，可使用适用于示例应用程序的其余部分的配置来解析依赖关系。代码清单18-25已将ProductTotalizer类声明为HomeController类的依赖项。

代码清单18-25　在Controllers文件夹下的HomeController.cs文

件中添加依赖关系

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {
        private IRepository repository;
        private ProductTotalizer totalizer;

        public HomeController(IRepository repo, Product
Totalizer total) {
            repository = repo;
            totalizer = total;
        }

        public ViewResult Index() {
            ViewBag.Total = totalizer.Total;
            return View(repository.Products);
        }
    }
}
```

Index操作方法会添加ViewBag属性，该属性包含ProductTotalizer类生成的产品总数，该类将显示在表格中，以查看本章开头部分添加到Index.cshtml文件中的ViewBag值。最后一步是告诉服务提供者

如何处理ProductTotalizer请求，如代码清单18-26所示。

代码清单18-26　在DependencyInjection文件夹下的Startup.cs文件中配置服务提供者

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IRepository, MemoryRe
pository>();
            services.AddTransient<IModelStorage, Dictio
naryStorage>();
            services.AddTransient<ProductTotalizer>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
```

```
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

在这种情况下，服务类型和实现类型之间不存在映射，所以需要有AddTransient扩展方法。该方法接收单个类型参数，该参数的值通知服务提供者应将ProductTotalizer类实例化以解析此类型的依赖项。

这种方法的优点是服务提供者将解析任何具体类声明的依赖项，而不是简单地实例化控制器中的具体类，因此更改配置以便使用专门的子类来解析具体类的依赖项。具体类由服务提供者管理，并且受生命周期功能的控制。如果运行示例应用程序，你将看到模型中的Product对象的总价格将显示出来，如图18-9所示。

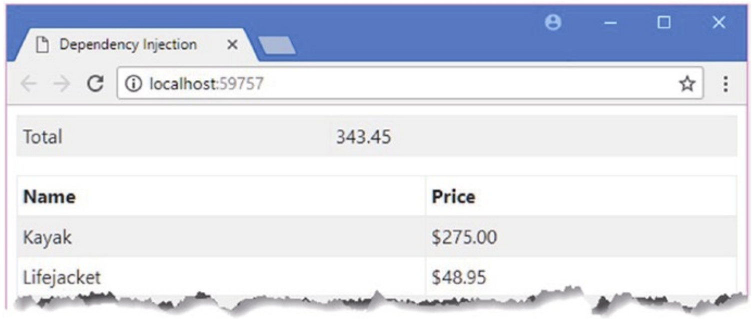


图18-9　对类使用依赖注入后的总价格

18.4　服务的生命周期

前面使用AddTransient扩展方法告诉服务提供者如何处理IRepository和IModelStorage接口上的依赖关系。AddTransient扩展方法是可以定义类型映射的4种不同方式之一。表18-3描述了用于告诉服务提供者如何解决依赖关系的扩展方法。表18-3中的扩展方法都使用类型参数，但也有可用于将类型对象作为参数接收的扩展方法，如果需要在运行时生成映射，就可以使用这些扩展方法。

表**18-3**　服务提供者的依赖注入扩展方法

扩展方法	描述
AddTransient<service,implType>()	告诉服务提供者为服务类型上的每个依赖关系创建实现类型的新实例
AddTransient<service>()	用于注册单个类型（将为每个依赖项进行实例化）
AddTransient<service>(factoryFunc)	用于注册将被调用的工厂模式，以便为服务类型的每个依赖项创建实现对象
AddScoped<service,implType>() AddScoped<service>() AddScoped<service>(factoryFunc)	这些扩展方法告诉服务提供者重用实现类型的实例，以便与公共作用域关联的组件所做的所有服务请求（通常是单个HTTP请求）共享同一对象。这些扩展方法遵循与相应的AddTransient扩展方法相同的模式
AddSingleton<service,implType>() AddSingleton<service>() AddSingleton<service(factoryFunc)	这些扩展方法告诉服务提供者为第一个服务请求创建实现类型的新实例，然后为每个后续服务请求重用
AddSingleton<service>(instance)	为服务提供者提供可应用于服务所有服务请求的对象，服务提供者将不会创建任何新对象

18.4.1　使用瞬态生命周期

开始使用依赖注入的最简单方法是使用AddTransient扩展方法，该方法告诉服务提供者在

需要解析依赖关系时创建实现类型的新实例。这是Startup类中已经存在的配置，如下所示：

```
...
public void ConfigureServices(IServiceCollection servic
es) {
    services.AddTransient<IRepository, MemoryRepository
>();
    services.AddTransient<IModelStorage, DictionaryStor
age>();
    services.AddTransient<ProductTotalizer>();
    services.AddMvc();
}
...
```

表18-3对生命周期做了介绍。瞬态生命周期会产生在每次解决依赖关系时创建实现类的新实例的成本，但优点在于不必担心管理并发访问或确保对象可以安全地重用于多个请求。

为了演示瞬态生命周期，在MemoryRepository类中重写ToString方法，以便生成全局唯一标识符（GUID），如代码清单18-27所示。

代码清单18-27　重写Models文件夹下的MemoryRepository.cs文件中的ToString方法

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
    public class MemoryRepository : IRepository {
        private IModelStorage storage;
        private string guid = System.Guid.NewGuid().ToS
tring();

        public MemoryRepository(IModelStorage modelStor
e) {
            storage = modelStore;
            new List<Product> {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M }
            }.ForEach(p => AddProduct(p));
        }
        public IEnumerable<Product> Products => storage
.Items;

        public Product this[string name] => storage[nam
e];

        public void AddProduct(Product product) =>
            storage[product.Name] = product;

        public void DeleteProduct(Product product) =>
            storage.RemoveItem(product.Name);
```

```
        public override string ToString() {
            return guid;
        }
    }
}
```

GUID使你可以轻松地识别MemoryRepository类的特定实例，并了解不同的生命周期方法如何改变服务提供者的行为方式。代码清单18-28更新了Home控制器上的Index操作方法，以便为通过存储库设置为GUID的View Bag创建Controller属性。

代码清单18-28　在Controllers文件夹下的HomeController.cs文件中使用View Bag

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {
        private IRepository repository;
        private ProductTotalizer totalizer;

        public HomeController(IRepository repo, Product
Totalizer total) {
```

```
            repository = repo;
            totalizer = total;
        }

        public ViewResult Index() {
            ViewBag.HomeController = repository.ToStrin
g();
            ViewBag.Totalizer = totalizer.Repository.To
String();
            return View(repository.Products);
        }
    }
}
```

Index 操作方法会将值添加到View Bag中，其中包含通过ProductTotalizer类的构造函数直接接收的repository对象的GUID，可以在运行应用程序时看到它们。两个GUID不同，因为服务提供者已使用AddTransient扩展方法进行了配置，这意味着将创建一个新的MemoryRepository对象来解决HomeController的依赖项和ProductTotalizer的另一个属性。瞬态生命周期的作用如图18-10所示。

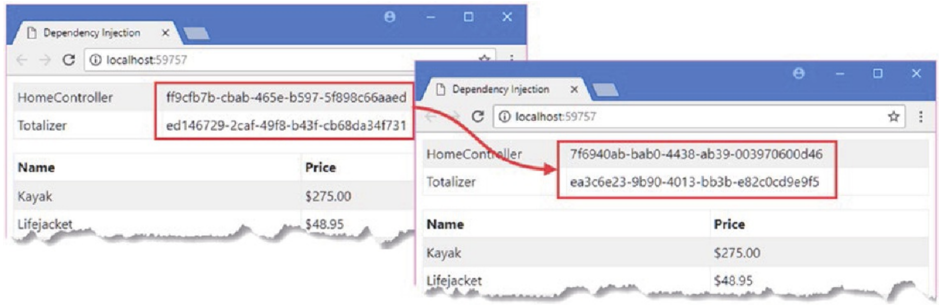


图18-10　瞬态生命周期的作用

每次重新加载Web页面时，新的HTTP请求都会导致MVC创建一个新的HomeController，这将导致创建两个新的MemoryRepository对象，每一个都有自己的GUID。

提　示

GUID是唯一的，或是接近唯一的。因此，当在自己的计算机上运行应用程序时，你将看到不同

的值。

使用工厂模式

AddTransient扩展方法的其中一个版本接收每次对服务类型有依赖性时调用的工厂模式。这允许创建的对象发生变化，以便不同的依赖项接收不同类型或实例的配置。代码清单18-29使用工厂模式根据应用程序运行的宿主环境来选择IRepository接口的不同实现。

代码清单18-29　在DependencyInjection文件夹下的Startup.cs文件中使用工厂模式

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {
        private IHostingEnvironment env;
        public Startup(IHostingEnvironment hostEnv) =>
env = hostEnv;

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddTransient<IRepository>(provider
 => {
                if (env.IsDevelopment()) {
                    var x = provider.GetService<MemoryR
epository>();
                    return x;
                } else {
                    return new AlternateRepository();
                }
            });
            services.AddTransient<MemoryRepository>();
            services.AddTransient<IModelStorage, Dictio
naryStorage>();
            services.AddTransient<ProductTotalizer>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
```

```
    }
}
```

第14章描述了ASP.NET如何为Startup类提供用来帮助设置应用程序的服务，包括用于确定宿主环境的IHostingEnvironment接口的实现。可以将这些服务作为参数传给Configure方法而不是ConfigureServices方法。为此，对Startup类添加一个构造函数，以提供对IHostingEnvironment对象的访问，并将其分配给一个名为env的字段。

在ConfigureServices方法中，使用AddTransient扩展方法定义一个使用了Lambda表达式的工厂模式。Lambda表达式接收一个System.IServiceProvider对象，该对象可用于创建使用表18-4所示方法向服务提供者注册的其他类型的实例。

表**18-4　IserviceProvider**的方法

方法	描述

GetService<service>()	使用服务提供者创建服务类型的新实例。如果请求的类型没有映射，就返回null
GetRequiredService<service>()	使用服务提供者创建服务类型的新实例。如果请求的类型没有映射，就抛出异常

在工厂模式中，使用IHostingEnvironment来确定应用程序是否在开发环境中运行。如果在开发环境中运行，就使用GetService方法创建MemoryRepository类的实例并从工厂返回作为用于IRepository依赖项的对象的函数。这里使用GetService方法创建对象，因为MemoryRepository在IModelStorage接口上有自己的依赖项，并使用服务提供者创建对象，这意味着检测和解决依赖关系将自动进行管理，但也意味着必须指定应用于MemoryRepository对象的生命周期，如下所示：

```
...
services.AddTransient<MemoryRepository>();
...
```

如果没有以上声明，服务提供者将不具备创建和管理MemoryRepository对象所需的信息。

如果应用程序未在开发环境中运行，那么factory函数将返回一个新的AlternateRepository实例。可以使用new关键字直接创建，因为不会在构造函数中声明任何依赖关系。

18.4.2　使用作用域的生命周期

作用域的生命周期会从实现类创建对象，用于解决与单个作用域关联的所有依赖项，通常表示单个HTTP请求（可以创建自己的作用域，但这在大多数应用程序中无效）。

由于默认作用域是HTTP请求，因此作用域的生命周期允许处理请求的所有组件共享单个对象，并且在编写自定义类（如路由）时，共享公共上下文数据通常非常有用。作用域的生命周期是通过使

用AddScoped扩展方法以配置服务提供者来创建的，如代码清单18-30所示。

提　示

如表18-4所示，还有其他接收工厂模式的AddScoped版本，可用于注册具体类型。这些方法的工作方式与前面演示的AddTransient方法相同，主要的区别在于它们创建的对象的生命周期不同。

代码清单18-30　在DependencyInjection文件夹下的Startup.cs文件中使用作用域的生命周期

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;
namespace DependencyInjection {
    public class Startup {
        private IHostingEnvironment env;

        public Startup(IHostingEnvironment hostEnv) =>
env = hostEnv;

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddScoped<IRepository, MemoryRepos
itory>();
            services.AddTransient<IModelStorage, Dictio
naryStorage>();
            services.AddTransient<ProductTotalizer>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

在示例应用程序中，HomeController和ProductTotalizer将一起实例化以处理请求，并且都要求存储库服务解决IRepository接口上的依赖关系。使用AddScoped方法可确保两个对象的依赖项都通过单个MemoryRepository对象来解决。可以通过运行示例应用程序来查看效果。浏览器显示的两个GUID是相同的，如图18-11所示。重新加载页面后，将创建一个新的HTTP请求，这意味着创建了一个新的MemoryRepository对象。

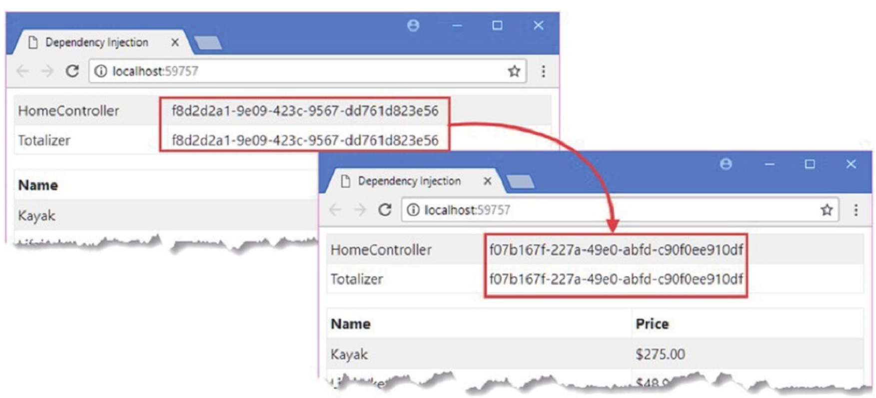


图18-11　作用域的生命周期产生的影响

18.4.3　使用单例生命周期

单例生命周期可确保单个对象用于解决给定服务类型的所有依赖项。使用单例生命周期时，必须确保用于解析依赖项的实现类对于并发访问是安全的。代码清单18-31更改了IRepository配置的作用域。

代码清单18-31　在DependencyInjection文件夹下的Startup.cs文件中使用作用域的生命周期

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
    public class Startup {
        private IHostingEnvironment env;

        public Startup(IHostingEnvironment hostEnv) =>
env = hostEnv;

        public void ConfigureServices(IServiceCollectio
```

```
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddTransient<IModelStorage, Dictio
naryStorage>();
            services.AddTransient<ProductTotalizer>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

AddSingleton方法在第一次创建MemoryRepository类的新实例时，必须在IRepository接口上解析依赖项，然后重用到任何后续依赖项，即使它们是不同的HTTP请求，如图18-12所示。

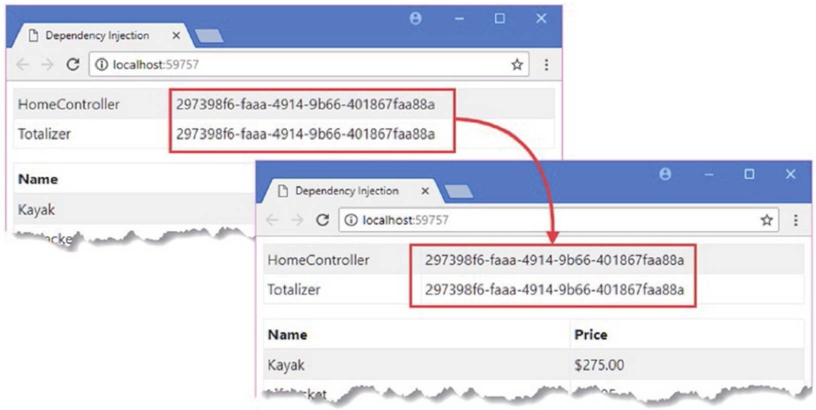


图18-12　单例生命周期的影响

18.5　使用操作注入

声明依赖关系的标准方法是使用构造函数，这是一种可以在任何类中使用的技术，依赖于依赖注入功能，依赖注入功能是ASP.NET Core平台的一部分。

MVC使用一种名为操作注入的替代方法来补充标准功能，允许通过参数将依赖关系声明为操作方法。严格说来，操作注入是由第26章介绍的模型绑定系统提供的，允许以不同的方式使用服务。可

使用FromServices属性执行操作注入，如代码清单18-32所示。

代码清单18-32　在Controllers文件夹下的HomeController.cs文件中使用操作注入

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index([FromServices]ProductTo
talizer totalizer) {
            ViewBag.HomeController = repository.ToStrin
g();
            ViewBag.Totalizer = totalizer.Repository.To
String();
            return View(repository.Products);
        }
    }
}
```

MVC使用服务提供者获取ProductTotalizer类的实例，并在调用Index操作方法时将其作为参数提供。操作注入相比构造函数注入更不常见，但是，当需要创建的对象具有依赖关系，并且仅在控制器定义的操作方法中需要时，操作注入会很有用。使用构造函数注入可以解决所有操作方法的依赖关系，即使用于处理请求的对象也不使用实现目标。使用FromServices属性修饰操作方法可缩小依赖项的焦点，并确保只有在需要时才实例化实现类型。

18.6　使用属性注入特性

第17章解释了如何通过声明属性并使用ControllerContext来修饰POCO控制器中的上下文数据。在阅读本章后，你就会明白这是依赖注入的一种特殊形式，称为属性注入。

MVC提供了一组专用属性，这些属性可用于

通过控制器中的属性注入和视图组件（参见第22章）接收特定类型。如果从控制器基类派生控制器，就不需要使用这些属性，因为上下文信息是通过convenience属性公开的，表18-5列出了可在POCO控制器中使用的属性。

表**18-5**　专用属性

名称	描述
ControllerContext	提供ActionContext类的功能超集
ActionContext	为操作方法提供上下文信息。控制器类会通过ActionContext属性公开上下文信息，第31章将介绍一组convenience属性
ViewContext	为视图操作提供上下文数据，包括标签助手
ViewComponentContext	为View组件设置ViewComponentContext属性
ViewDataDictionary	设置ViewDataDictionary属性以提供对模型绑定数据的访问

18.7　手动请求实现对象

ASP.NET的依赖注入功能以及MVC为属性注入和action注入提供的附加属性，提供了为大多数应用程序创建松散耦合组件所需的所有支持。但是有些时候，在不使用依赖注入的情况下获取接口的实现也是有用的。在这些情况下，可以直接与服务提供者一起工作，如代码清单18-33所示。

代码清单18-33　在Controllers文件夹下的HomeController.cs文件中直接使用服务提供者

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;
using Microsoft.Extensions.DependencyInjection;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {

        public ViewResult Index([FromServices]ProductTo
talizer totalizer) {

            IRepository repository =
                HttpContext.RequestServices.GetService<
IRepository>();

            ViewBag.HomeController = repository.ToStrin
```

```
g();
            ViewBag.Totalizer = totalizer.Repository.To
String();
            return View(repository.Products);
        }
    }
}
```

由同名属性返回的HttpContext对象定义了RequestServices方法，该方法返回一个IServiceProvider对象，可以调用表18-4中描述的扩展方法。以上代码删除了使用属性注入设置的Repository属性，并使用HttpContext.RequestServices属性获取IRepository接口的实现。这就是服务定位器模式，一些开发者认为应该避免使用这种模式。Mark Seemann在其博客上详细描述了这种模式可能导致的问题。当通过构造函数接收依赖关系的常规技术由于某种原因不能使用时，以这种方式获取服务是完全合理的。

18.8　小结

本章介绍了依赖注入在MVC应用程序中扮演的角色，依赖注入有助于创建松散耦合的组件，以便能够轻松地进行替换和隔离测试。此外，本章还演示了ASP.NET依赖注入功能和MVC为将依赖项注入属性和操作方法而提供的属性，介绍了配置服务提供者时可用的不同生命周期选项，并解释了它们如何影响对象的创建方式。下一章将介绍过滤器，它会在请求处理过程中添加额外的逻辑。

第19章　过滤器

过滤器用于向MVC请求处理注入额外的逻辑。它们提供了一种简单而优雅的方式来实现交叉关注。所谓交叉关注（cross-cutting concern），是指可用于整个应用程序，但又不适合放置在某个局部位置的功能，否则会打破关注点分离原则。典型的交叉关注例子是日志记录、授权和缓存。本章将展示MVC支持的不同类别的过滤器、如何创建和使用自定义过滤器以及如何控制它们的执行。

表19-1展示了上下文中使用的过滤器。

表**19-1**　上下文中使用的过滤器

问题	答案
什么是过滤器？	过滤器可以将逻辑应用于操作方法，而不必在控制器类中添加代码

过滤器有什么作用？	过滤器允许应用不属于操作的经典MVC模式定义的代码。可以实现更简单的控制器类和可重用的功能，并且可在整个应用程序中应用
如何使用过滤器？	MVC可通过不同的方式使用不同类型的过滤器。创建过滤器的最常见方法是创建一个类，该类将MVC为所需的筛选类型提供的属性作为子类
过滤器有什么陷阱或限制吗？	不同类型的过滤器提供的功能有重叠，有时候很难选择需要哪种类型的过滤器
除了过滤器还有其他的选择吗？	没有，过滤器是MVC的核心功能，用于实现日常所需的功能，比如用户授权

表19-2展示了本章要介绍的操作。

表**19-2**　本章要介绍的操作

操作	方法	代码清单
在请求处理中注入额外的逻辑	在控制器或操作方法上应用过滤器	代码清单19-6～代码清单19-9
限制对操作的访问	使用授权过滤器	代码清单19-10和代码清单19-11

将通用逻辑注入请求处理过程	使用操作过滤器	代码清单19-12～代码清单19-14
检查或改变操作方法产生的结果	使用结果过滤器	代码清单19-15～代码清单19-19
处理错误	使用异常过滤器	代码清单19-20和代码清单19-21
在过滤器中使用服务	声明过滤器构造函数中的依赖关系，在Startup类中注册服务，并使用TypeFilter属性应用过滤器	代码清单19-22～代码清单19-26
将过滤器置于生命周期管理之下	使用依赖注入的生命周期在Startup类中注册过滤器，并使用ServiceFilter属性应用过滤器	代码清单19-27～代码清单19-29
对应用程序中的每个操作方法应用过滤器	使用全局过滤器	代码清单19-30～代码清单19-32
更改执行过滤器的顺序	使用Order参数	代码清单19-33～代码清单19-36

19.1　准备示例项目

在本章中，将遵循前几章使用的相同方法来创建示例应用程序。使用ASP.NET Core Web Application（.NET Core）模板创建一个名为Filters的Empty项目。代码清单19-1显示了对Startup类所做的更改，用于启用MVC框架和开发所需的其他中间件。

代码清单19-1　Filters文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Filters {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
```

```
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

19.1.1　启用SSL

本章中的一些示例需要使用SSL，默认情况下SSL是禁用的。要启用SSL，请从Visual Studio菜单中选择Filter Properties，然后在Debug选项卡中勾选“Enable SSL https://localhost:44318/”复选框，如图19-1所示。请注意记录分配的端口，这对于每个项目将是不同的。

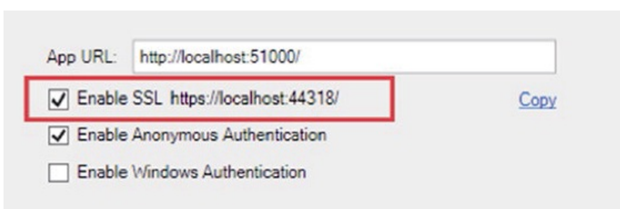


图19-1　启用SSL

19.1.2　创建控制器和视图

本章中的控制器很简单，因为本章的重点是应用程序的其他位置。这里创建Controllers文件夹，添加一个名为HomeController.cs的类文件，内容如代码清单19-2所示。

代码清单19-2 Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");
    }
}
```

Index操作方法将呈现一个名为Message的视图，并将一个字符串传递给该视图。这里创建Views/Shared文件夹，并添加一个名为Message.cshtml的Razor视图文件，内容如代码清单

19-3所示。

代码清单19-3　Views/Shared文件夹下的Message.cshtml文件的内容

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Filters</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @if (Model is string) {
        @Model
    } else if (Model is IDictionary<string, string>) {
        var dict = Model as IDictionary<string, string>
;
        <table class="table table-sm table-striped tabl
e-bordered">
            <thead><tr><th>Name</th><th>Value</th></tr>
</thead>
            <tbody>
                @foreach (var kvp in dict) {
                    <tr><td>@kvp.Key</td><td>@kvp.Value
</td></tr>
                }
            </tbody>
```

```
        </table>
    }
</body>
</html>
```

Message视图是弱类型视图，将显示字符串或字典Dictionary<string,string>，在当前情况下，会显示一个表格。

Message视图依赖于Bootstrap CSS包来为HTML元素设置样式。要将Bootstrap包添加到项目中，可以使用Bower Configuration File模板，在项目的根文件夹中创建bower.json文件，并将Bootstrap包添加到dependencies部分，如代码清单19-4所示。

代码清单19-4　在Filters文件夹下的bower.json文件中添加Bootstrap包中

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
```

```
}
```

最后的准备工作是在Views文件夹中创建_ViewImports.cshtml文件，用于设置内置的标签助手，以便在Razor视图中使用，如代码清单19-5所示。

代码清单19-5　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

如果运行示例应用程序，看到的输出将如图19-2所示。

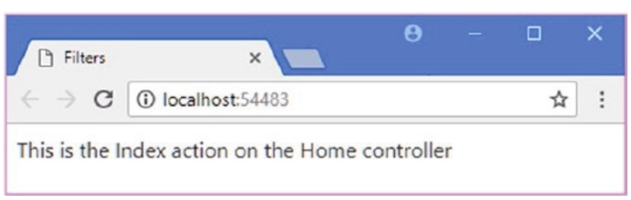


图19-2　运行示例应用程序的输出

提　示

系统可能会提示需要信任Visual Studio生成的证书。请接受建议，这与本章中的示例依赖于SSL有关。

19.2　使用过滤器

过滤器允许从控制器中删除在操作方法中应用的一些逻辑，并在可重用类中重新定义。例如，假设要确保只能使用 HTTPS 访问操作方法，而不能使用常规的未加密 HTTP。HttpRequest 上下文对象提供了是否使用HTTPS的信息，如代码清单19-6所示。

代码清单19-6　在Controllers文件夹下的HomeController.cs文件中验证是否使用HTTPS

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace Filters.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            if (!Request.IsHttps) {
                return new StatusCodeResult(StatusCodes
.Status403Forbidden);
            } else {
                return View("Message",
                    "This is the Index action on the Ho
me controller");
            }
        }
    }
}
```

这演示了如何在没有使用过滤器的情况下处理HTTPS问题。如果运行示例应用程序，浏览器将请求Index操作方法并处理不带HTTPS的默认URL，还会返回一个StatusCodeResult，用以在响应中发送HTTP 403状态编码（如第17章所述）。如果要求带

HTTPS的默认URL（比如https://localhost:44318），Index操作方法将通过渲染Message视图进行响应（可能需要在浏览器显示结果之前确认安全警告）。图19-3显示了这两种结果。

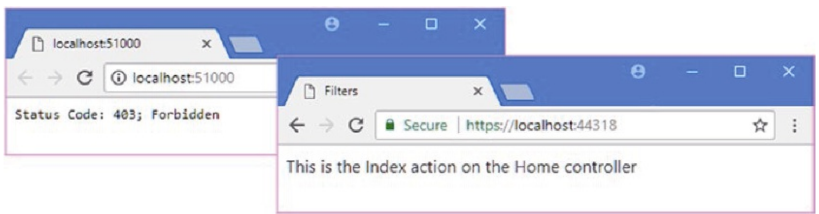


图19-3　限制对HTTPS请求的访问的结果



提　示

如果没有从示例中获得期望的结果，请清除浏览器的历史记录。浏览器通常会拒绝向已经生成SSL错误的服务器发送请求，这是很好的安全设

置，但在开发过程中可能不是很方便。

代码清单19-6中的代码可以正常工作，但也存在一些问题。其中一个问题是，操作方法包含的代码更多的是实现安全策略，而不是处理请求、更新模型和选择响应。另一个更严重的问题是，在操作方法中包括的HTTP检测代码不能很好地进行扩展，并且必须在控制器中的每个操作方法中重复，如代码清单19-7所示。

代码清单19-7　在Controllers文件夹下的HomeController.cs文件中添加一个操作方法

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
```

```
            if (!Request.IsHttps) {
                return new StatusCodeResult(StatusCodes
.Status403Forbidden);
            } else {
                return View("Message",
                    "This is the Index action on the Ho
me controller");
            }
        }

        public IActionResult SecondAction() {
            if (!Request.IsHttps) {
                return new StatusCodeResult(StatusCodes
.Status403Forbidden);
            } else {
                return View("Message",
                    "This is the SecondAction action on
 the Home controller");
            }
        }
    }
}
```

必须记住，在每个需要HTTPS的控制器的每个操作方法中都要执行相同的检查。实现安全策略的代码是控制器的重要组成部分，虽然看起来实现很简单，但会使控制器更难理解，忘记将它们添加到新的操作方法中只是时间问题，这是安全漏洞。这个问题可以通过使用过滤器来解决，如代码清单

19-8所示。

代码清单19-8　在Controllers文件夹下的HomeController.cs文件中使用过滤器

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    public class HomeController : Controller {

        [RequireHttps]
        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        [RequireHttps]
        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");
    }
}
```

RequireHttps特性会将一个内置的过滤器应用于HomeController类。这将限制对操作方法的访问，仅支持HTTPS请求，并允许从每个方法中删除

安全代码，使方法可以专注于处理成功接收到的请求。

注　意

RequireHttps过滤器的工作方式与代码清单19-7中的自定义代码完全相同。对于GET请求，RequireHttps特性将客户端重定向到原始请求的URL，但是要求通过使用HTTPS方案来执行操作，因此对http://localhost/Home/Index的请求会被重定向到https://localhost/Home/Index。这对于部署的大多数应用程序来说是有意义的，但在开发过程中不是这样，因为HTTP和HTTPS使用的是不同的端口。RequireHttpsAttribute类定义了一个名为

HandleNonHttpsRequest的受保护方法，可以重写该方法以更改其行为。也可参考19.4节，从头开始创建原始的功能。

当然，还需要将RequireHttps特性应用到每个操作方法，有时候你可能会忘记这件事。但是过滤器有如下很有用的技巧：将特性应用于控制器类与将特性应用于每个单独的操作方法具有相同的效果，如代码清单19-9所示。

代码清单19-9　将过滤器应用于HomeController.cs文件中的所有操作方法

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    [RequireHttps]
    public class HomeController : Controller {
```

```
        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");
    }
}
```

过滤器有不同的级别。如果要限制访问某些操作而不影响其他操作，可以将RequireHttps特性应用于这些方法。如果要保护所有操作方法，包括未来添加到控制器中的任何操作，那么可以将RequireHttps特性应用于控制器类。如果要对应用程序中的每个操作应用过滤器，可以使用全局过滤器，这将在本章后面进行介绍。

19.3 实现过滤器

你已经看到如何使用过滤器，现在介绍过滤器在幕后是如何实现的。过滤器实现了IFilterMetadata

接口，该接口定义在Microsoft.AspNetCore.Mvc.Filters命名空间中：

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IFilterMetadata { }
}
```

该接口是空的，不需要过滤器类来实现任何特定的行为。这是因为有几种不同类型的过滤器，它们的工作方式各不相同，并用于不同的目的。

表19-3列出了各种类型的过滤器、定义它们的接口以及它们所做的事情（MVC还支持一些其他类型的过滤器，但它们不是直接使用的，而是集成到功能中，并通过特定属性进行应用，包括第20章将要介绍的Produces和Consumes属性。）

表**19-3**　不同类型的过滤器

过滤器	接口	描述

授权过滤器	IAuthorizationFilter IAsyncAuthorizationFilter	使用应用程序的安全策略，包括用户授权
操作过滤器	IActionFilter IAsyncActionFilter	在执行操作方法之前或之后立即执行工作
结果过滤器	IResultFilter IAsyncResultFilter	在处理操作方法的结果之前或之后立即执行工作
异常过滤器	IExceptionFilter IAsyncExceptionFilter	处理异常

表19-3中的描述有些模糊，因为可以使用过滤器进行很多工作，仅受想象力和需要解决的问题的限制。如果你了解过滤器是如何工作的，这将变得更清楚，现在有两个要点需要理解。

首先，表19-3中的每种类型的过滤器都有两种不同的接口。过滤器可以同步或异步地执行它们的工作，比如，同步实现的过滤器实现了IResultFilter接口，而异步实现的过滤器则实现了IAsyncResultFilter接口。

其次，过滤器按特定的顺序执行。首先执行授权过滤器，然后执行文件操作，接下来执行结果过滤器。只有抛出异常才会执行异常过滤器，这会打乱正常的执行序列。

获取上下文数据

过滤器以FilterContext对象的形式提供上下文数据。FilterContext类派生自ActionContext，ActionContext也是第17章描述的ControllerContext类的基类。为方便起见，表19-4列出了从ActionContext类继承的属性以及FilterContext定义的附加属性。

表**19-4　FilterContext**类的属性

属性	描述
ActionDescriptor	返回一个ActionDescriptor对象，该对象描述了操作方法

HttpContext	返回一个HttpContext对象，该对象提供HTTP请求的详细信息以及将要作为回复发送的HTTP响应
ModelState	返回一个ModelStateDictionary对象，用于验证客户端发送的数据
RouteData	返回一个RouteData对象，描述路由系统处理请求的方式
Filters	返回已应用于操作方法的过滤器列表，表示为IList <IFilterMetadata>

19.4　使用授权过滤器

授权过滤器用于实现应用程序的安全策略。授权过滤器在其他类型的过滤器之前和执行操作方法之前执行。以下是IAuthorizationFilter接口的定义：

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAuthorizationFilter : IFilterMetadata {

        void OnAuthorization(AuthorizationFilterContext context);
    }
}
```

可调用OnAuthorization方法来为过滤器提供验

证授权请求。对于异步授权过滤器，以下是IAsyncAuthorizationFilter接口的定义：

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncAuthorizationFilter : IFilterMetadata {

        Task OnAuthorizationAsync(AuthorizationFilterContext context);
    }
}
```

调用OnAuthorizationAsync方法，以便过滤器可以授权请求。无论使用哪个接口，过滤器都会通过AuthorizationFilterContext对象（从FilterContext类派生）接收描述请求的上下文数据，并添加一个重要属性——Result，如表19-5所示。

表**19-5　AuthorizationFilterContext**类的属性

属性	描述

Result	当请求不符合应用程序的授权策略时，IActionResult属性由授权过滤器设置，如果设置了这个属性，MVC将会呈现IActionResult而不是调用操作方法

创建授权过滤器

为了演示授权过滤器的工作原理，在示例项目中创建Infrastructure文件夹，并添加一个名为HttpsOnlyAttribute.cs的类文件（见代码清单19-10），用它定义过滤器。

代码清单19-10　Infrastructure文件夹下的HttpsOnlyAttribute.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters.Infrastructure {
    public class HttpsOnlyAttribute : Attribute, IAutho
rizationFilter {

        public void OnAuthorization(AuthorizationFilter
Context context) {
            if (!context.HttpContext.Request.IsHttps) {
                context.Result =
                    new StatusCodeResult(StatusCodes.St
atus403Forbidden);
```

```
            }
        }
    }
}
```

如果请求符合授权策略，则授权过滤器不执行任何操作，MVC会进入下一个过滤器，最后执行操作方法。

注　意

在之前的MVC中，可以用于限制特定用户和用户组访问的Authorize属性是使用过滤器实现的，但ASP.NET Core MVC中不再是这样。Authorize属性仍然使用，但工作方式不同。在幕后，全局过滤器（本章后面将介绍全局过滤器）用于检测Authorize属性并执行ASP.NET Core Identity系统定

义的策略，但Authorize属性不再是过滤器，不能实现IAuthorizationFilter接口。第29章将介绍如何使用ASP.NET Core Identity系统和Authorize属性。

如果存在问题，则过滤器会设置传递给OnAuthorization方法的AuthorizationFilterContext对象的Result属性。这可以阻止事件的进一步执行，并为MVC提供返回到客户端的结果。在以上代码中，HttpsOnlyAttribute类检查HttpRequest Context对象的IsHttps属性，并将Result属性设置为在没有HTTPS的情况下发出请求时中断执行。代码清单19-11展示了如何在Home控制器中使用新的过滤器。

代码清单19-11　在Controllers文件夹下的HomeController.cs文件中使用自定义过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");
    }
}
```

这个自定义过滤器重新实现了代码清单19-7中的操作方法的功能。在实际项目中，与执行重定向（如内置的RequireHttps过滤器）相比，这并没有提供更多的功能，因为用户不理解403状态码的含义，但这个示例对于演示授权过滤器如何工作十分有用。

过滤器的单元测试

过滤器的单元测试的大部分工作是设置传递给过滤器方法的Context对象。所需的模仿数据取决于过滤器使用的上下文信息。例如，下面是用于代码清单19-10中的HttpsOnly过滤器的单元测试。

```
using System.Linq;
using Filters.Infrastructure;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Abstractions;
using Microsoft.AspNetCore.Mvc.Filters;
using Moq;
using Xunit;

namespace Tests {

    public class FilterTests {

        [Fact]
        public void TestHttpsFilter() {

            // Arrange
            var httpRequest = new Mock<HttpRequest>();
            httpRequest.SetupSequence(m => m.IsHttps).R
eturns(true)
                                                     .R
eturns(false);
```

```
            var httpContext = new Mock<HttpContext>();
            httpContext.SetupGet(m => m.Request).Return
s(httpRequest.Object);

            var actionContext = new ActionContext(httpC
ontext.Object,
                new Microsoft.AspNetCore.Routing.RouteD
ata(),
                new ActionDescriptor());
            var authContext = new AuthorizationFilterCo
ntext(actionContext,
                Enumerable.Empty<IFilterMetadata>().ToL
ist());

            HttpsOnlyAttribute filter = new HttpsOnlyAt
tribute();

            // Act and Assert
            filter.OnAuthorization(authContext);
            Assert.Null(authContext.Result);

            filter.OnAuthorization(authContext);
            Assert.IsType(typeof(StatusCodeResult), aut
hContext.Result);
            Assert.Equal(StatusCodes.Status403Forbidden
,
                (authContext.Result as StatusCodeResult
).StatusCode);
        }
    }
}
```

首先模仿HttpRequest和HttpContext Context对

象，这样就可以使用或不使用HTTPS呈现请求。如果想测试这两个条件，可以这样做：

```
...
httpRequest.SetupSequence(m => m.IsHttps).Returns(true).
Returns(false);
...
```

以上语句设置了HttpRequest.IsHttps属性，使它返回一系列测试值：该属性在第一次读取时返回true，并在第二次读取时返回false。一旦有了一个HttpContext对象，就可以用它创建一个ActionContext对象，从而创建用于执行单元测试的AuthorizationContext对象。通过检查AuthorizationFilterContext对象的Result属性，可以测试过滤器如何响应非HTTPS请求，然后测试HTTP请求发生的情况。设置AuthorizationFilterContext对象时需要很多类型，并且它们依赖于许多ASP.NET Core和MVC命名空

间，但是一旦有了Context对象，那么编写其余的测试就会比较简单。

19.5　使用操作过滤器

了解操作过滤器的最佳方式是查看它们的定义。以下是IActionFilter接口：

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IActionFilter : IFilterMetadata {

        void OnActionExecuting(ActionExecutingContext context);

        void OnActionExecuted(ActionExecutedContext context);
    }
}
```

当操作过滤器已应用于操作方法时，可在调用操作方法之前调用OnActionExecuting方法，之后再

调用OnActionExecuted方法。操作过滤器通过两个不同的上下文类提供了上下文数据，它们分别是OnActionExecuting方法的ActionExecutingContext类和OnActionExecuted方法的ActionExecutedContext类。这两个上下文类都扩展了FilterContext类，详见表19-4。

ActionExecutingContext类用于描述将要调用的操作，表19-6描述了该类的属性。

表**19-6　ActionExecutingContext**类的属性

属性	描述
Controller	返回要调用操作方法的控制器（操作方法的详细信息可通过从基类继承的 ActionDescriptor 属性获得）
ActionArguments	返回将传递给操作方法的参数字典（按名称索引），过滤器可以插入、移除或更改参数
Result	如果过滤器为Result属性分配IActionResult，请求进程将被跳过，并且操作结果将用于在不调用操作方法的情况下生成返回给客户端的响应

ActionExecutedContext类用于表示已执行的操作，其中定义了表19-7描述的属性。

表**19-7　ActionExecutedContext**类的属性

属性	描述
Controller	返回将调用操作方法的控制器
Canceled	如果另一个操作过滤器通过将操作结果分配给ActionExecutingContext对象的Result属性来将请求处理过程短路，就将这个属性设置为true
Exception	包含由操作方法抛出的所有异常
ExceptionDispatchInfo	返回一个ExceptionDispatchInfo对象，其中包含由操作方法抛出的任何异常的堆栈跟踪详细信息
ExceptionHandled	若设置为true，表示过滤器已处理异常，异常不会再进一步传播
Result	返回由操作方法返回的IActionResult。如果需要，过滤器可以更改或替换操作结果

19.5.1　创建操作过滤器

操作过滤器是一种通用工具，可用于实现应用程序中的任何横切问题。操作过滤器可用于在调用操作之前中断请求进程，并可在执行操作后更改结果。创建操作过滤器的最简单方法是从ActionFilterAttribute类派生一个类，并让它实现IActionFilter接口。为了演示，在Infrastructure文件夹中添加一个名为ProfileAttribute.cs的类文件，内容如代码清单19-12所示。

代码清单19-12　Infrastructure文件夹下的ProfileAttribute.cs文件的内容

```
using System.Diagnostics;
using System.Text;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters.Infrastructure {

    public class ProfileAttribute : ActionFilterAttribu
te {
        private Stopwatch timer;

        public override void OnActionExecuting(ActionEx
ecutingContext context) {
            timer = Stopwatch.StartNew();
        }
```

```
        public override void OnActionExecuted(ActionExe
cutedContext context) {
            timer.Stop();
            string result = "<div>Elapsed time: "
                + $"{timer.Elapsed.TotalMilliseconds} m
s</div>";
            byte[] bytes = Encoding.ASCII.GetBytes(resu
lt);
            context.HttpContext.Response.Body.Write(byt
es, 0, bytes.Length);
        }
    }
}
```

以上代码使用Stopwatch对象来测量通过在OnActionExecuting方法中启动计时器来执行操作方法所需的毫秒数，并在OnActionExecuted方法中停止计数。为了观察结果，可使用Context对象获取HttpResponse对象，并在响应中包含简单的HTML片段。

代码清单19-13显示了如何在Home控制器上应用Profile特性（这里删除了以前的过滤器，以便可以接收HTTP请求）。

提　示

有一种很特别的情况，控制器也是操作过滤器。控制器基类实现了IActionFilter和IAsyncActionFilter接口，这意味着可以通过重写这些接口定义的方法来创建操作过滤器功能。对于POCO控制器，MVC检查这些类并检查它们是否实现了其中一个操作过滤器接口，并自动将它们作为操作过滤器使用。

代码清单19-13　在Controllers文件夹下的HomeController.cs文件中应用过滤器

```
using Microsoft.AspNetCore.Mvc;
```

```
using Filters.Infrastructure;

namespace Filters.Controllers {

    [Profile]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");
    }
}
```

如果运行示例应用程序，你将看到图19-4所示的结果。你看到的毫秒数与开发时使用的计算机的性能有关。

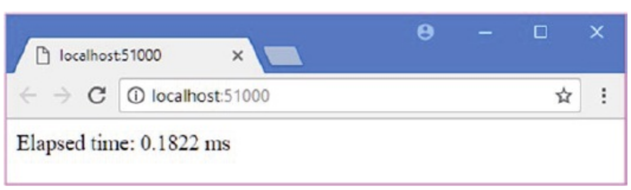


图19-4　运行结果

注　意

将HTML片段直接写入响应依赖于浏览器能否支持显示格式不正确的HTML文档。你在过滤器中生成的div元素出现在响应正文的开头，在Razor视图生成表示HTML文档开头的DOCTYPE和html元素之前。这种技术可以用于生成诊断信息，但是不应该在生产环境使用。

19.5.2　创建异步操作过滤器

IAsyncActionFilter接口用于定义异步操作的操作过滤器，定义如下：

```
using System.Threading.Tasks;
```

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncActionFilter : IFilterMetada
ta {

        Task OnActionExecutionAsync(ActionExecutingCont
ext context,
            ActionExecutionDelegate next);
    }
}
```

在执行操作方法前后，存在依赖于任务延续的单一方法，该方法允许过滤器运行。代码清单19-14显示了如何在Profile过滤器中使用OnActionExecutionAsync方法。

代码清单19-14　在Infrastructure文件夹下的ProfileAttribute.cs文件中创建异步操作过滤器

```
using System.Diagnostics;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters.Infrastructure {

    public class ProfileAttribute : ActionFilterAttribu
te {

        public override async Task OnActionExecutionAsy
```

```
nc(
                ActionExecutingContext context,
                ActionExecutionDelegate next) {

            Stopwatch timer = Stopwatch.StartNew();

            await next();

            timer.Stop();
            string result = "<div>Elapsed time: "
                + $"{timer.Elapsed.TotalMilliseconds} m
s</div>";
            byte[] bytes = Encoding.ASCII.GetBytes(resu
lt);
            await context.HttpContext.Response.Body.Wri
teAsync(bytes,
                0, bytes.Length);
        }
    }
}
```

ActionExecutingContext对象为过滤器提供了上下文数据，ActionExectionDelegate对象表示要执行的操作方法（或下一个过滤器）。过滤器在调用委托之前做准备工作，然后在委托完成后完成这些工作。代理会返回一个Task对象，因而上面的代码中使用了await关键字。

19.6　使用结果过滤器

结果过滤器在MVC处理操作方法返回的操作结果的前后应用。结果过滤器能够更改或替换操作结果或完全取消请求（即使已调用操作方法）。下面是定义结果过滤器的IResultFilter接口：

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IResultFilter : IFilterMetadata {

        void OnResultExecuting(ResultExecutingContext c
ontext);

        void OnResultExecuted(ResultExecutedContext con
text);
    }
}
```

结果过滤器遵循与操作过滤器相同的模式。在处理操作方法产生的操作结果之前调用OnResultExecuting 方法，并通过ResultExecutingContext对象提供上下文信息。

ResultExecutingContext类是从FilterContext派生的，并定义了表19-8中的属性。

表**19-8　ResultExecutingContext**类定义的属性

属性	描述
Controller	返回执行操作方法的控制器
Cancel	将这个属性设置为true，就会停止处理操作结果以生成响应
Result	返回由操作方法返回的IActionResult对象

调用OnResultExecuted方法后，MVC已经处理了操作结果，并通过ResultExecutedContext类的实例提供了上下文数据，ResultExecutedContext类继承自FilterContext，并定义了表19-9中的属性。

表**19-9　ResultExecutedContext**类定义的属性

属性	描述

Controller	返回执行操作方法的控制器
Canceled	这个属性指示请求是否被取消
Exception	包含由操作方法抛出的所有异常
ExceptionDispatchInfo	返回一个ExceptionDispatchInfo对象，其中包含由操作方法抛出的任何异常的堆栈跟踪详细信息
ExceptionHandled	将这个属性设置为true表示过滤器已处理异常，异常不会再进一步传播
Result	返回用于生成对客户端响应的IActionResult对象

19.6.1　创建结果过滤器

ResultFilterAttribute类实现了结果过滤器接口，并提供了创建结果过滤器的最简单方法。为了演示结果过滤器的工作原理，将一个名为ViewResultDetailsAttribute.cs的类文件添加到Infrastructure文件夹中，内容如代码清单19-15所示。

代码清单19-15　Infrastructure文件夹下的ViewResultDetailsAttribute.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {

    public class ViewResultDetailsAttribute : ResultFil
terAttribute {

        public override void OnResultExecuting(ResultEx
ecutingContext context) {
            Dictionary<string, string> dict = new Dicti
onary<string, string> {
                ["Result Type"] = context.Result.GetTyp
e().Name,
            };

            ViewResult vr;
            if ((vr = context.Result as ViewResult) !=
null) {
                dict["View Name"] = vr.ViewName;
                dict["Model Type"] = vr.ViewData.Model.
GetType().Name;
                dict["Model Data"] = vr.ViewData.Model.
ToString();
            }

            context.Result = new ViewResult {
                ViewName = "Message",
```

```
                ViewData = new ViewDataDictionary(
                        new EmptyModelMetadataProvider(
),
                        new ModelStateDictionary()) { M
odel = dict }
                };
        }
    }
}
```

这个类只重写了OnResultExecuting方法，并使用Context对象更改了用于发送响应到客户端的操作结果。过滤器创建了一个ViewResult对象，该对象使用包含简单诊断信息的字典作为视图模型传递给Message视图。

OnResultExecuting方法在操作方法生成了操作结果之后，但在处理生成的结果之前调用，并且更改了Context对象的Result对象的值，从而允许为应用过滤器的操作方法提供不同类型的结果。代码清单19-16显示了应用于Home控制器的结果过滤器。

代码清单19-16　在Controllers文件夹下的HomeController.cs文

件中使用结果过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

    [ViewResultDetails]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");
    }
}
```

如果运行示例应用程序，你将看到结果过滤器的效果，如图19-5所示。

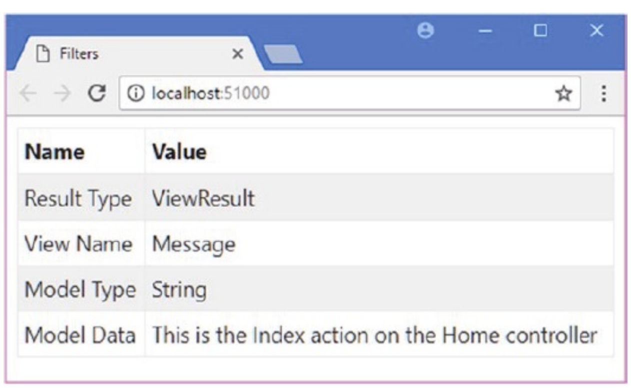


图19-5　结果过滤器的效果

19.6.2　创建异步结果过滤器

IAsyncResultFilter接口可用于创建异步结果过滤器，定义如下：

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncResultFilter : IFilterMetada
ta {

        Task OnResultExecutionAsync(ResultExecutingCont
ext context,
            ResultExecutionDelegate next);
    }
}
```

这个接口类似于异步操作过滤器。代码清单19-17 重写了 ViewResultDetailsAttribute 类以实现IAsyncResultFilter接口。

代码清单19-17　在Infrastructure文件夹下的

ViewResultDetailsAttribute.cs文件中创建异步结果过滤器

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {
    public class ViewResultDetailsAttribute : ResultFil
terAttribute {

        public override async Task OnResultExecutionAsy
nc(
                ResultExecutingContext context,
                ResultExecutionDelegate next) {

            Dictionary<string, string> dict = new Dicti
onary<string, string> {
                ["Result Type"] = context.Result.GetTyp
e().Name,
            };

            ViewResult vr;
            if ((vr = context.Result as ViewResult) !=
null) {
                dict["View Name"] = vr.ViewName;
                dict["Model Type"] = vr.ViewData.Model.
GetType().Name;
                dict["Model Data"] = vr.ViewData.Model.
ToString();
            }
```

```
            context.Result = new ViewResult {
                ViewName = "Message",
                ViewData = new ViewDataDictionary(
                        new EmptyModelMetadataProvider(
),
                        new ModelStateDictionary()) {
                            Model = dict
                        }
                };

            await next();
        }
    }
}
```

请注意，需要负责调用作为OnResultExecutionAsync方法的参数接收的委托。如果不调用委托，请求处理管道将不会完成，并且不会呈现操作结果。

19.6.3 创建混合操作/结果过滤器

区分请求处理的操作阶段和结果阶段并不总是有帮助。这可能是因为你希望将两个阶段视为一个步骤，或是因为你希望过滤器影响响应执行操作的

方式，而不是通过干预结果来实现。因此，能够创建既可以是操作过滤器又可以是结果过滤器，并且能够在每个阶段执行工作的过滤器是非常有用的。

以下要求很常见：要求ActionFilterAttribute类实现两种类型的过滤器接口，这意味着可以在单个属性中混合和匹配过滤器类型。为了演示这是如何工作的，修改代码清单19-18中的ProfileAttribute类的代码，以便将操作过滤器与结果过滤器相结合。

代码清单19-18　在Infrastructure文件夹下的ProfileAttribute.cs文件中创建混合过滤器

```
using System.Diagnostics;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters.Infrastructure {

    public class ProfileAttribute : ActionFilterAttribu
te {
        private Stopwatch timer;
        private double actionTime;

        public override async Task OnActionExecutionAsy
```

```
nc(
                ActionExecutingContext context,
                ActionExecutionDelegate next) {

            timer = Stopwatch.StartNew();

            await next();

            actionTime = timer.Elapsed.TotalMillisecond
s;
        }

        public override async Task OnResultExecutionAsy
nc(
                ResultExecutingContext context,
                ResultExecutionDelegate next) {

            await next();

            timer.Stop();
            string result = "<div>Action time: "
                + $"{actionTime} ms</div><div>Total tim
e: "
                + $"{timer.Elapsed.TotalMilliseconds} m
s</div>";
            byte[] bytes = Encoding.ASCII.GetBytes(resu
lt);

            await context.HttpContext.Response.Body.Wri
teAsync(bytes,
                0, bytes.Length);
        }
    }
}
```

这里对两种类型的过滤器使用了异步方法，但也可以混合使用它们，从而获取所需的功能，因为这些方法的默认实现会调用它们的同步对应版本。在过滤器中，可使用Stopwatch来测量要处理的操作所需的时间，以及总的经历时间，并将结果写入响应。代码清单19-19已将混合过滤器应用于Home控制器。

代码清单19-19　在Controllers文件夹下的HomeController.cs文件中使用混合过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

    [Profile]
    [ViewResultDetails]
    public class HomeController : Controller {
        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
```

```
e controller");
    }
}
```

如果运行示例应用程序，你将看到类似于图19-6的输出。输出显示在ViewResultDetails提供的内容之后，因为是在结果过滤器的最后处理阶段写入的，而不是来自先前版本中使用的操作过滤器。

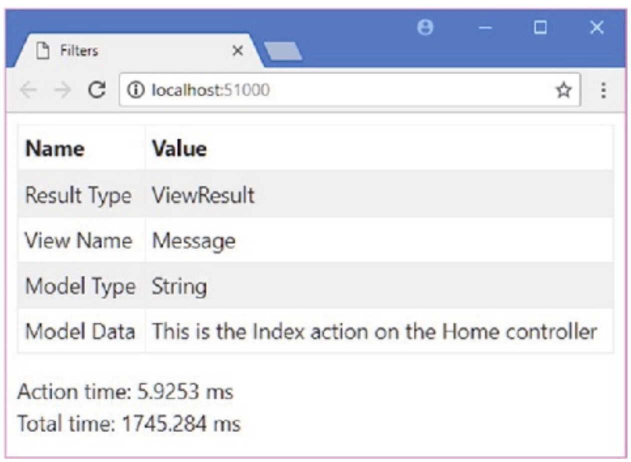


图19-6　混合操作/结果过滤器的输出

19.7　使用异常过滤器

异常过滤器允许响应异常，而无须在每个操作方法中写入try … catch代码块。异常过滤器可以应

用于控制器类或操作方法。当操作方法或已应用于操作方法的操作过滤器或结果过滤器未处理异常时，将会调用它们（操作过滤器和结果过滤器可以通过将Context对象的ExceptionHandled属性设置为true来处理未处理的异常）。异常过滤器实现了IExceptionFilter接口，定义如下：

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IExceptionFilter : IFilterMetadata
 {

        void OnException(ExceptionContext context);
    }
}
```

如果遇到未处理的异常，则调用OnException方法。IAsyncExceptionFilter接口可用于创建异步异常过滤器，如果需要使用异步API响应异常，这将非常有用。以下是异步接口的定义：

```
using System.Threading.Tasks;

```

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncExceptionFilter : IFilterMetadata {

        Task OnExceptionAsync(ExceptionContext context);
    }
}
```

OnExceptionAsync方法是来自IExceptionFilter接口的OnException方法的异步对象，当存在未处理的异常时调用。

IAsyncExceptionFilter接口和IExceptionFilter接口都通过 ExceptionContext 类提供上下文数据，该类派生自FilterContext，并定义了表19-10中描述的属性。

表**19-10　ExceptionContext**类定义的属性

属性	描述
Exception	包含抛出的任意异常

ExceptionDispatchInfo	返回一个ExceptionDispatchInfo对象，该对象包含异常的栈跟踪详细信息
ExceptionHandled	这个属性用于指示异常是否已被处理
Result	设置将用于生成响应的IActionResult

创建异常过滤器

ExceptionFilterAttribute类实现了两个异常过滤器接口，也是创建过滤器的最简单方法，可以应用为属性。异常过滤器的常见用途是为特定异常类型提供自定义错误页面，以便为用户提供相比标准错误处理功能所能提供的更有用的信息。例如，将一个名为RangeExceptionAttribute.cs的类文件添加到Infrastructure文件夹中，文件内容如代码清单19-20所示。

代码清单19-20　Infrastructure文件夹下的RangeExceptionAttribute.cs文件的内容

理异常的类型，如果类型为

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {
    public class RangeExceptionAttribute : ExceptionFil
terAttribute {

        public override void OnException(ExceptionConte
xt context) {
            if (context.Exception is ArgumentOutOfRange
Exception) {
                context.Result = new ViewResult() {
                    ViewName = "Message",
                    ViewData = new ViewDataDictionary(
                        new EmptyModelMetadataProvider(
),
                        new ModelStateDictionary()) {
                            Model = @"The data received
 by the
                                 application cannot be
processed"
                    }
                };
            }
        }
    }
}
```

这个过滤器使用ExceptionContext对象获取未处理异常的类型，如果类型为

ArgumentOutOfRangeException，则会创建用于向用户显示消息的操作结果。代码清单19-21向Home控制器添加一个操作方法，并将异常过滤器应用到这个操作方法。

代码清单19-21　在Controllers文件夹下的HomeController.cs文件中应用异常过滤器

```
using Filters.Infrastructure;
using Microsoft.AspNetCore.Mvc;
using System;

namespace Filters.Controllers {

    [Profile]
    [ViewResultDetails]
    [RangeException]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");

        public ViewResult GenerateException(int? id) {
```

```
            if (id == null) {
                throw new ArgumentNullException(nameof(
id));
            } else if (id > 10) {
                throw new ArgumentOutOfRangeException(n
ameof(id));
            } else {
                return View("Message", $"The value is {
id}");
            }
        }
    }
}
```

GenerateException操作方法依赖默认路由模式从请求URL中接收可空的int值。如果值为null，操作方法将抛出ArgumentNullException异常；如果值大于50，则抛出ArgumentOutOfRangeException异常。如果有值且在指定范围内，那么操作方法返回一个ViewResult对象。

可以通过运行示例应用程序并请求URL——/Home/GenerateException/100来测试异常过滤器。最后一段已超出操作方法的预期范围，这将抛

出由过滤器处理的异常类型，产生图19-7所示的结果。如果请求URL——/Home/ GenerateException，那么由操作方法抛出的异常将不会被过滤器处理，并且将使用默认错误处理。

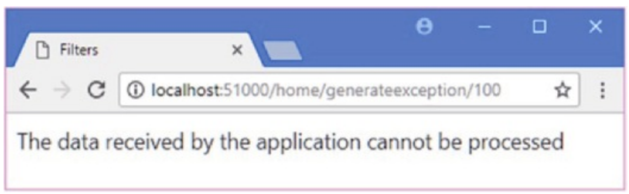


图19-7　使用异常过滤器

19.8　为过滤器使用依赖注入

当从convenience属性类（如ExceptionFilterAttribute）派生过滤器时，MVC将创建过滤器类的新实例来处理每个请求。这是一种合理的方法，因为可以避免任何可能的重用或并发问题，并且满足了开发人员需要的大多数过滤器类的需求。

另一种方法是使用依赖注入系统为过滤器选择不同的生命周期。在过滤器中使用依赖注入有两种不同的方法，下面进行介绍。

19.8.1　解决过滤器依赖项

第一种方法是使用依赖注入来管理过滤器的上下文数据，以允许不同类型的过滤器共享数据，或者为单个过滤器共享数据，而实例本身用于处理其他请求。为了演示这是如何工作的，在Infrastructure文件夹中添加一个名为FilterDiagnostics.cs的类文件，内容如代码清单19-22所示。

代码清单19-22　Infrastructure文件夹下的FilterDiagnostics.cs文件的内容

```
using System.Collections.Generic;

namespace Filters.Infrastructure {
```

```
    public interface IFilterDiagnostics {
        IEnumerable<string> Messages { get; }
        void AddMessage(string message);
    }
    public class DefaultFilterDiagnostics : IFilterDiag
nostics {
        private List<string> messages = new List<string
>();

        public IEnumerable<string> Messages => messages
;

        public void AddMessage(string message) =>
            messages.Add(message);
    }
}
```

IFilterDiagnostics接口定义了一个简单的模型，用于在筛选过程中收集诊断消息。这里将使用DefaultFilterDiagnostics类来实现。代码清单19-23更新了Startup类，以使用新的接口及其实现来配置服务提供者。

代码清单19-23　在Filters文件夹下的Startup.cs文件中配置服务提供者

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddScoped<IFilterDiagnostics, Defa
ultFilterDiagnostics>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

以上代码使用AddScoped扩展方法来配置服务提供者，这意味着所有实例化的处理单个请求的过滤器都将收到相同的DefaultFilterDiagnostics对象。这是在过滤器之间共享自定义上下文数据的基础。

1．创建具有依赖关系的过滤器

下一步是创建在IFilterDiagnostics接口上声明依赖关系的过滤器。在Infrastructure文件夹中创建一个名为TimeFilter.cs的类文件，内容如代码清单19-24所示。

代码清单19-24　Infrastructure文件夹下的TimeFilter.cs文件的内容

```
using System.Diagnostics;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

    public class TimeFilter : IAsyncActionFilter, IAsyncResultFilter {
        private Stopwatch timer;
        private IFilterDiagnostics diagnostics;

        public TimeFilter(IFilterDiagnostics diags) {
            diagnostics = diags;
        }

        public async Task OnActionExecutionAsync(
                ActionExecutingContext context,
                ActionExecutionDelegate next) {
```

```
            timer = Stopwatch.StartNew();
            await next();
            diagnostics.AddMessage($@"Action time:
                {timer.Elapsed.TotalMilliseconds}");
        }

        public async Task OnResultExecutionAsync(
                ResultExecutingContext context,
                ResultExecutionDelegate next) {

            await next();
            timer.Stop();
            diagnostics.AddMessage($@"Result time:
                {timer.Elapsed.TotalMilliseconds}");
        }
    }
}
```

TimeFilter类是混合操作/结果过滤器，可从先前的示例重新创建计时器功能，但使用IFilterDiagnostics接口存储定时信息，该接口被声明为构造函数参数，并在创建过滤器时由依赖注入系统提供。

请注意，TimeFilter类直接实现了过滤器接口，而不是从convenience属性类派生。正如你将看

到的，依赖于依赖注入的过滤器是通过不同的属性应用的，并且不用于直接修饰控制器或action。

为了演示过滤器如何使用依赖项注入来共享上下文数据，在Infrastructure文件夹中添加一个名为DiagnosticsFilter.cs的类文件，内容如代码清单19-25所示。

代码清单19-25　Infrastructure文件夹下的DiagnosticsFilter.cs文件的内容

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

    public class DiagnosticsFilter : IAsyncResultFilter
 {
        private IFilterDiagnostics diagnostics;

        public DiagnosticsFilter(IFilterDiagnostics dia
gs) {
            diagnostics = diags;
        }

        public async Task OnResultExecutionAsync(
```

```
                ResultExecutingContext context,
                ResultExecutionDelegate next) {

            await next();

            foreach (string message in diagnostics?.Mes
sages) {
                byte[] bytes = Encoding.ASCII
                    .GetBytes($"<div>{message}</div>");
                await context.HttpContext.Response.Body
                    .WriteAsync(bytes, 0, bytes.Length)
;
            }
        }
    }
}
```

DiagnosticsFilter类是结果过滤器，可将IFilterDiagnostics接口的实现作为构造函数参数接收，并将其中包含的消息写入响应。

2．应用过滤器

最后一步是将过滤器应用于控制器类。标准的C#属性不具有解析构造函数依赖项的整体能力，这就是前面的过滤器不是属性的原因。相反，而是应

用TypeFilter特性，并使用所需的过滤器类型进行配置，如代码清单19-26所示。

代码清单19-26　在Controllers文件夹下的HomeController.cs文件中应用使用依赖注入的过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;
using System;

namespace Filters.Controllers {

    [TypeFilter(typeof(DiagnosticsFilter))]
    [TypeFilter(typeof(TimeFilter))]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");

        public ViewResult GenerateException(int? id) {
            if (id == null) {
                throw new ArgumentNullException(nameof(
id));
            } else if (id > 10) {
                throw new ArgumentOutOfRangeException(n
```

```
ameof(id));
            } else {
                return View("Message", $"The value is {
id}");
            }
        }
    }
}
```

提　示

代码清单19-26中过滤器的应用顺序很重要，正如19.10节中解释的那样。

TypeFilter特性为每个请求创建过滤器类的新实例，但使用依赖注入功能，这样可以创建松散耦合的组件，并将用于解决依赖关系的对象置于生命周

期管理之下。

在该例中，这意味着代码清单19-26中应用的两个过滤器都将接收到相同的IFilterDiagnostics实现对象，因此TimeFilter类写入的消息将被DiagnosticsFilter类写入响应。图 19-8 显示了启动应用程序并请求应用程序的默认URL后可以看到的效果。

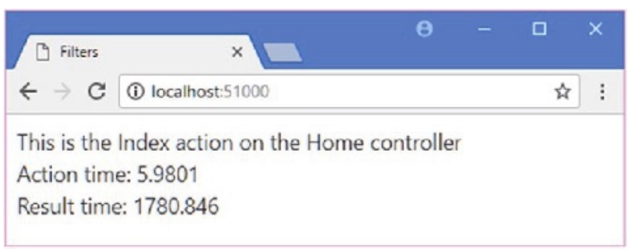


图19-8　启动应用程序并请求应用程序的默认URL的效果

19.8.2　管理过滤器的生命周期

当使用TypeFilter特性时，会为每个请求创建过滤器类的新实例。这与将过滤器作为属性直接应用的行为相同，只是TypeFilter特性允许过滤器类声明

通过服务提供者解析的依赖项。

相比使用ServiceFilter特性更进一步，可使用服务提供者创建过滤器对象。这使得过滤器对象也可以置于生命周期管理之下。作为演示，代码清单19-27修改了TimeFilter类，以便保持简单的平均时间值。

代码清单19-27　在Infrastructure文件夹下TimeFilter.cs文件中保持简单的平均时间值

```
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

    public class TimeFilter : IAsyncActionFilter, IAsyn
cResultFilter {
        private ConcurrentQueue<double> actionTimes = n
ew ConcurrentQueue<double>();
        private ConcurrentQueue<double> resultTimes = n
ew ConcurrentQueue<double>();
        private IFilterDiagnostics diagnostics;

```

```
        public TimeFilter(IFilterDiagnostics diags) {
            diagnostics = diags;
        }

        public async Task OnActionExecutionAsync(
                ActionExecutingContext context, ActionE
xecutionDelegate next) {

            Stopwatch timer = Stopwatch.StartNew();
            await next();
            timer.Stop();
            actionTimes.Enqueue(timer.Elapsed.TotalMill
iseconds);
            diagnostics.AddMessage($@"Action time:
                {timer.Elapsed.TotalMilliseconds}
                Average: {actionTimes.Average():F2}");
        }

        public async Task OnResultExecutionAsync(
                ResultExecutingContext context, ResultE
xecutionDelegate next) {

            Stopwatch timer = Stopwatch.StartNew();
            await next();
            timer.Stop();
            resultTimes.Enqueue(timer.Elapsed.TotalMill
iseconds);
            diagnostics.AddMessage($@"Result time:
                {timer.Elapsed.TotalMilliseconds}
                Average: {resultTimes.Average():F2}");
        }
    }
}
```

过滤器现在使用线程安全的集合来存储为请求处理的操作阶段和结果阶段记录的时间，并在每次处理请求时使用单独的Stopwatch。代码清单19-28已经在Startup类中将TimeFilter类注册为服务提供者的单例。

代码清单19-28　在Filters文件夹下的Startup.cs文件中配置服务提供者

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IFilterDiagnostics, D
efaultFilterDiagnostics>();
            services.AddSingleton<TimeFilter>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

请注意，这里还改变了IFilterDiagnostics的生命周期，它是一个单例。如果继续为每个请求创建一个新的实例，那么单例TimeFilter将从DiagnosticsFilter接收一个不同的IFilterDiagnostics对象，该对象继续通过TypeFilter特性进行实例化，并为每个请求创建一个。

应用过滤器

使用ServiceType特性将过滤器应用于控制器，如代码清单19-29所示。

代码清单19-29　在Controllers文件夹下的HomeController.cs文

}

件中使用过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;
using System;

namespace Filters.Controllers {
    [TypeFilter(typeof(DiagnosticsFilter))]
    [ServiceFilter(typeof(TimeFilter))]
    public class HomeController : Controller {

        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");

        public ViewResult SecondAction() => View("Messa
ge",
            "This is the SecondAction action on the Hom
e controller");

        public ViewResult GenerateException(int? id) {
            if (id == null) {
                throw new ArgumentNullException(nameof(
id));
            } else if (id > 10) {
                throw new ArgumentOutOfRangeException(n
ameof(id));
            } else {
                return View("Message", $"The value is {
id}");
            }
        }
    }
}
```

可以通过运行示例应用程序并请求默认URL来查看效果。因为IFilterDiagnostics接口的单个实现对象用于解析所有依赖关系，所以显示的消息集合会随着每个请求而建立，如图19-9所示。

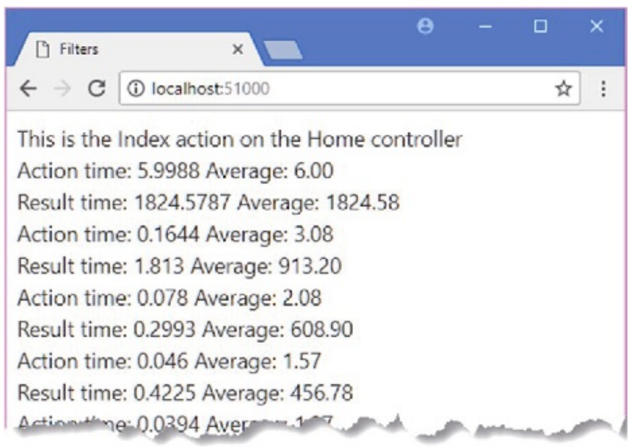


图19-9　使用service provider管理过滤器的生命周期

19.9　创建全局过滤器

你可以将过滤器应用于控制器类，而不必将它们应用于各个操作方法。全局过滤器更进一步，只在Startup类中应用一次，就可自动应用于应用程序的每个控制器中的每个操作方法。任何过滤器都可

以用作全局过滤器，为了演示，创建一个名为ViewResultDiagnostics.cs的类文件并保存到Infrastructure文件夹中，文件的内容如代码清单19-30所示。

代码清单19-30　Infrastructure文件夹下的ViewResultDiagnostics.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {
    public class ViewResultDiagnostics : IActionFilter
{
        private IFilterDiagnostics diagnostics;

        public ViewResultDiagnostics(IFilterDiagnostics
 diags) {
            diagnostics = diags;
        }

        public void OnActionExecuting(ActionExecutingCo
ntext context) {
            // do nothing - not used in this filter
        }

        public void OnActionExecuted(ActionExecutedCont
ext context) {
            ViewResult vr;
```

```
            if ((vr = context.Result as ViewResult) !=
null) {
                diagnostics.AddMessage($"View name: {vr
.ViewName}");
                diagnostics.AddMessage($@"Model type:
                    {vr.ViewData.Model.GetType().Name}"
);
            }
        }
    }
}
```

这个过滤器使用IFilterDiagnostics对象来存储有关ViewResult action结果的视图名称和模型类型方面的消息。代码清单19-31将这个过滤器与DiagnosticsFilter类一起应用于全局，过滤器依赖于DiagnosticsFilter类写入诊断消息。

代码清单19-31　在Filters文件夹下的Startup.cs文件中注册全局过滤器

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;
namespace Filters {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddScoped<IFilterDiagnostics, Defa
ultFilterDiagnostics>();
            services.AddScoped<TimeFilter>();
            services.AddScoped<ViewResultDiagnostics>();

            services.AddScoped<DiagnosticsFilter>();
            services.AddMvc().AddMvcOptions(options => {

                options.Filters.AddService(typeof(ViewR
esultDiagnostics));
                options.Filters.AddService(typeof(Diagn
osticsFilter));
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

全局过滤器是通过配置MVC服务包来设置的。本例使用了全局过滤器的

MvcOptions.Filters.AddService方法。AddService方法接收将使用ConfigureServices方法中其他位置指定的生命周期规则实例化的.NET类型。可将其他过滤器类型的生命周期更改为范围受限，以便为每个请求创建新的实例。因此，将会为每个控制器的每个请求创建并应用ViewResultDiagnostics和DiagnosticsFilter过滤器的新实例。

提　示

还可以使用Add方法而不是AddService方法添加全局过滤器，这样就可以将过滤器对象注册为全局过滤器，而不依赖于依赖注入和服务提供者。

将一个名为GlobalController.cs的类文件添加到Controllers文件夹中，内容如代码清单19-32所示。

代码清单19-32　Controllers文件夹下的GlobalController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    public class GlobalController : Controller {

        public ViewResult Index() => View("Message",
            "This is the global controller");
    }
}
```

没有将任何过滤器应用于Global控制器，但如果启动应用程序并请求/global，你将看到两个全局过滤器的输出，如图19-10所示。

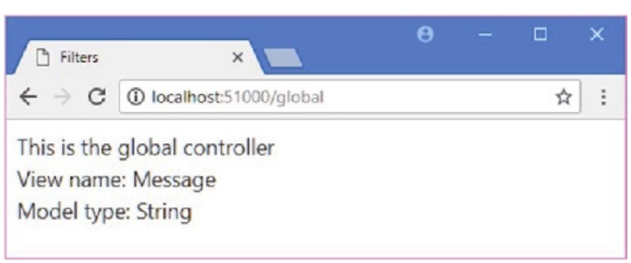


图19-10　两个全局过滤器的输出

19.10　理解和更改过滤器的执行顺序

过滤器以特定顺序运行，即授权过滤器→操作过滤器→结果过滤器。但是，如果有多个给定类型的过滤器，则它们的应用顺序由已应用过滤器的范围驱动。为了演示这如何工作，将一个名为MessageAttribute.cs的类文件添加到Infrastructure文件夹中，文件内容如代码清单19-33所示。

代码清单19-33　Infrastructure文件夹下的MessageAttribute.cs文件的内容

```
using System.Text;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

    public class MessageAttribute : ResultFilterAttribu
te {
        private string message;

        public MessageAttribute(string msg) {
            message = msg;
```

```
        }

        public override void OnResultExecuting(ResultEx
ecutingContext context) {
            WriteMessage(context, $"<div>Before Result:
{message}</div>");
        }

        public override void OnResultExecuted(ResultExe
cutedContext context) {
            WriteMessage(context, $"<div>After Result:{
message}</div>");
        }

        private void WriteMessage(FilterContext context
, string msg) {
            byte[] bytes = Encoding.ASCII
                .GetBytes($"<div>{msg}</div>");
            context.HttpContext.Response
                .Body.Write(bytes, 0, bytes.Length);
        }
    }
}
```

这是一个结果过滤器，但并非在处理操作结果前后将HTML片段写入响应。过滤器写入的消息是通过构造函数参数来配置的，该参数在应用为属性时可以使用。代码清单19-34简化了Home控制器，并将以前示例中的过滤器替换为Message过滤器的

多个实例。

代码清单19-34　在Controllers文件夹下的HomeController.cs文件中使用过滤器

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

    [Message("This is the Controller-Scoped Filter")]
    public class HomeController : Controller {

        [Message("This is the First Action-Scoped Filte
r")]
        [Message("This is the Second Action-Scoped Filt
er")]
        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");
    }
}
```

上面已经修改了一组全局过滤器，以便在这里使用Message过滤器，如代码清单19-35所示。

代码清单19-35　在Filters文件夹下的Startup.cs文件中创建全局过滤器

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddScoped<IFilterDiagnostics, Defa
ultFilterDiagnostics>();
            services.AddScoped<TimeFilter>();
            services.AddScoped<ViewResultDiagnostics>()
;
            services.AddScoped<DiagnosticsFilter>();
            services.AddMvc().AddMvcOptions(options =>
{
                options.Filters.Add(new
                    MessageAttribute("This is the Globa
lly-Scoped Filter"));
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
```

```
    }
}
```

当Index操作方法响应请求时，将使用4个过滤器实例。如果运行应用程序并请求默认URL，你将在浏览器中看到以下输出：

```
Before Result:This is the Globally-Scoped Filter
Before Result:This is the Controller-Scoped Filter
Before Result:This is the First Action-Scoped Filter
Before Result:This is the Second Action-Scoped Filter
After Result:This is the Second Action-Scoped Filter
After Result:This is the First Action-Scoped Filter
After Result:This is the Controller-Scoped Filter
After Result:This is the Globally-Scoped Filter
```

默认情况下，MVC 运行全局过滤器，然后将过滤器应用于控制器，最后将过滤器应用于操作方法。一旦调用操作方法或处理操作结果，过滤器的栈就被释放了，因而输出中的After Result消息以相反的顺序显示。

修改过滤器的执行顺序

可以通过实现IOrderedFilter接口来更改默认顺序，MVC在指定如何按顺序堆叠过滤器的过程中会查找这个接口。这个接口的定义如下：

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IOrderedFilter : IFilterMetadata {
        int Order { get; }
    }
}
```

Order属性会返回一个int值，较低的值将通知MVC在执行具有较高顺序值的筛选之前应用过滤器。convenience属性已经实现了IOrder值，代码清单19-36为应用于Home控制器的过滤器设置了Order属性。

提　示

TypeFilter和ServiceFilter特性还实现了IOrderedFilter接口，这意味着可以在使用依赖注入时更改过滤器的执行顺序。

代码清单19-36　在Controllers文件夹下的HomeController.cs文件中设置过滤器的执行顺序

```
using Filters.Infrastructure;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

    [Message("This is the Controller-Scoped Filter", Or
der = 10)]
    public class HomeController : Controller {

        [Message("This is the First Action-Scoped Filte
r", Order = 1)]
        [Message("This is the Second Action-Scoped Filt
er", Order = -1)]
        public ViewResult Index() => View("Message",
            "This is the Index action on the Home contr
oller");
    }
```

```
}
```

过滤器的Order值也可以为负数，这是确保在具有默认执行顺序的任何全局过滤器之前应用过滤器的有用方法（尽管也可以在创建全局过滤器时设置执行顺序）。如果运行示例应用程序，你将看到消息的输出顺序已更改，以反映新设置的优先级。

```
Before Result:This is the Second Action-Scoped Filter
Before Result:This is the Globally-Scoped Filter
Before Result:This is the First Action-Scoped Filter
Before Result:This is the Controller-Scoped Filter
After Result:This is the Controller-Scoped Filter
After Result:This is the First Action-Scoped Filter
After Result:This is the Globally-Scoped Filter
After Result:This is the Second Action-Scoped Filter
```

19.11　小结

本章讲述了如何将交叉关注问题的逻辑封装为过滤器，展示了各种不同的过滤器以及它们的实现方法。此外，本章还讨论了如何将过滤器作为属性应用于控制器和操作方法，以及如何将它们作为全

局过滤器来应用。下一章将展示如何使用控制器来创建Web服务。

第20章　API控制器

并非所有控制器都用于向客户端发送HTML文档，还有API控制器用于提供对应用程序数据的访问。这是以前通过单独的Web API框架提供的功能，但现在已经集成到ASP.NET Core MVC中。本章将解释API控制器在Web应用程序中的作用，描述它们解决的问题，并演示如何创建、测试和使用它们。表20-1描述了API控制器的背景。

表**20-1**　**API**控制器的背景

问题	答案
什么是API控制器？	API控制器与常规控制器相似，除了它们的操作方法产生的响应是发送到客户端的数据对象而不是HTML标记之外
API控制器有什么作用？	API控制器允许客户端访问应用程序中的数据，而不会收到将内容呈现给用户所需的HTML标记。并不是所有客户端都是浏览器，也并不是所有客户端都向用户显示数据。API控制器使应用程序变成开放的，可以支持新类型的客户端或由第三方开发的客户端

如何使用API控制器？	API控制器可像常规HTML控制器一样使用
使用API控制器时有哪些陷阱或局限性？	最常见的问题涉及数据对象被序列化以发送到客户端的方式。有关详细信息，请参阅20.4节
是否有其他的替代方案？	不必在项目中使用API控制器，但这样做可以增加平台对客户端的价值

表20-2列出了本章要介绍的操作。

表**20-2**　本章要介绍的操作

操作	方法	代码清单
提供对应用程序数据的访问	创建API控制器	代码清单20-10
从API控制器请求数据	使用Ajax查询，直接使用浏览器API或通过库，比如jQuery	代码清单 20-11～代码清单20-13

覆盖内容协商过程	使用Produces属性	代码清单 20-14～代码清单20-16
允许客户端通过URL中指定的数据格式覆盖Accept标头	在Startup类中添加格式化映射，添加捕获数据格式的片段变量，并可选择应用FormatFilter特性	代码清单20-17和代码清单20-18
为内容协商过程提供全面支持	启用HttpNotAcceptableOutputFormatter格式化程序并设置RespectBrowserAcceptHeader配置属性	代码清单20-19和代码清单20-20
使用不同的操作方法接收不同格式的数据	使用Consumes属性	代码清单20-21

20.1　准备示例项目

对于本章，使用ASP.NET Core Web Application（.NET Core）模板创建一个名为ApiControllers的Empty项目。

20.1.1　创建模型和存储库

首先创建Models文件夹，添加一个名为

Reservation.cs的类文件，并用它定义如代码清单20-1所示的模型类。

代码清单20-1　Models文件夹下的Reservation.cs文件的内容

```
namespace ApiControllers.Models {

    public class Reservation {
        public int ReservationId { get; set; }
        public string ClientName { get; set; }
        public string Location { get; set; }
    }
}
```

这里还将一个名为IRepository.cs的文件添加到Models文件夹中，并用它定义模型存储库的接口，如代码清单20-2所示。

代码清单20-2　Models文件夹下的IRepository.cs文件的内容

```
using System.Collections.Generic;

namespace ApiControllers.Models {

    public interface IRepository {

        IEnumerable<Reservation> Reservations { get; }
```

```
        Reservation this[int id] { get; }

        Reservation AddReservation(Reservation reservat
ion);
        Reservation UpdateReservation(Reservation reser
vation);
        void DeleteReservation(int id);
    }
}
```

添加一个名为MemoryRepository.cs的类文件到Models文件夹中，并用它定义IRepository接口的非持久性实现，如代码清单20-3所示。

代码清单20-3　Models文件夹下的MemoryRepository.cs文件的内容

```
using System.Collections.Generic;

namespace ApiControllers.Models {

    public class MemoryRepository : IRepository {
        private Dictionary<int, Reservation> items;

        public MemoryRepository() {
            items = new Dictionary<int, Reservation>();
            new List<Reservation> {
                new Reservation { ClientName = "Alice",
 Location = "Board Room" },
                new Reservation { ClientName = "Bob", L
```

```
ocation = "Lecture Hall" },
                new Reservation { ClientName = "Joe", L
ocation = "Meeting Room 1" }
            }.ForEach(r => AddReservation(r));
        }

        public Reservation this[int id] => items.Contai
nsKey(id) ? items[id] : null;

        public IEnumerable<Reservation> Reservations =>
 items.Values;

        public Reservation AddReservation(Reservation r
eservation) {
            if (reservation.ReservationId == 0) {
                int key = items.Count;
                while (items.ContainsKey(key)) { key++;
 };
                reservation.ReservationId = key;
            }
            items[reservation.ReservationId] = reservat
ion;
            return reservation;
        }

        public void DeleteReservation(int id) => items.
Remove(id);

        public Reservation UpdateReservation(Reservatio
n reservation)
            => AddReservation(reservation);

    }
}
```

存储库在实例化时会创建一组简单的模型对象，并且由于没有持久存储，因此当应用程序停止或重新启动时，任何更改都将丢失。有关如何在SportsStore示例应用程序中创建持久存储库的示例，请参阅第8章。

20.1.2　创建控制器和视图

在本章的后面部分，将创建REST控制器，但在准备时，需要创建常规控制器，从而为以后的例子提供基础。创建Controllers文件夹，并添加一个名为HomeController.cs的文件，Home控制器的定义如代码清单20-4所示。

代码清单20-4　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {
```

```
    public class HomeController : Controller {
        private IRepository repository { get; set; }

        public HomeController(IRepository repo) => repo
sitory = repo;

        public ViewResult Index() => View(repository.Re
servations);

        [HttpPost]
        public IActionResult AddReservation(Reservation
 reservation) {
            repository.AddReservation(reservation);
            return RedirectToAction("Index");
        }
    }
}
```

Home控制器定义了Index操作，这是应用程序的默认值，用于渲染数据模型。另外，还定义了AddReservation操作，但只能用于接收HTTP POST请求，并且将用于从用户接收表单数据。这些操作遵循第17章中描述的Post/Redirect/Get模式，这样重新加载网页时就不会创建重复提交的表单。

创建布局，以便可以将HTML内容与HTML文

档标头分离出来，这将简化本章稍后部分的一些更改。创建Views/Shared文件夹，添加一个名为_Layout.cshtml的布局文件，并添加代码清单20-5所示的标记。

代码清单20-5　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RESTful Controllers</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

接下来，创建Views/Home文件夹，添加一个名为Index.cshtml的视图文件，并添加代码清单20-6所示的内容。

代码清单20-6　Views/Home文件夹下的Views.cshtml文件的内容

```
@model IEnumerable<Reservation>
@{  Layout = "_Layout"; }

<form id="addform" asp-action="AddReservation" method="
post">
    <div class="form-group">
        <label for="ClientName">Name:</label>
        <input class="form-control" name="ClientName" /
>
    </div>
    <div class="form-group">
        <label for="Location">Location:</label>
        <input class="form-control" name="Location" />
    </div>
    <div class="text-center panel-body">
        <button type="submit" class="btn btn-sm btn-pri
mary">Add</button>
    </div>
</form>

<table class="table table-sm table-striped table-border
ed m-2">
    <thead><tr><th>ID</th><th>Client</th><th>Location</
th></tr></thead>
    <tbody>
        @foreach (var r in Model) {
            <tr>
                <td>@r.ReservationId</td>
                <td>@r.ClientName</td>
                <td>@r.Location</td>
            </tr>
```

```
        }
    </tbody>
</table>
```

这种强类型视图接收一系列Reservation对象作为模型，并使用Razor foreach循环来填充表格。另外，还有一个表单，已配置为将POST请求发送到AddReservation操作。

本章中的示例依赖于Bootstrap CSS包。要将Bootstrap包添加到项目中，可使用Bower Configuration File模板在项目的根目录中创建bower.json文件，并将Bootstrap包添加到dependencies部分，如代码清单20-7所示。

代码清单20-7　在bower.json文件中添加Bootstrap包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

接下来，在Views文件夹中创建_ViewImports.cshtml文件，并在其中设置用于Razor视图的内置标签助手，然后导入模型命名空间，如代码清单20-8所示。

代码清单20-8　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using ApiControllers.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

为了启用开发所需的MVC框架和中间件，对Startup类进行修改，如代码清单20-9所示。另外，还使用AddSingleton方法为模型存储库设置了服务映射。

代码清单20-9　在ApiControllers文件夹下的Startup.cs文件中启用中间件

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

设置HTTP端口

本章中的一些示例需要通过手动输入URL进行测试。为了描述更容易，下面设置用于接收HTTP

请求的端口。从Visual Studio的Project菜单中选择ApiControllers Properties，显示Debug选项卡，将App URL的值更改为http://localhost:7000/，如图20-1所示。确保在设置端口号后保存更改。

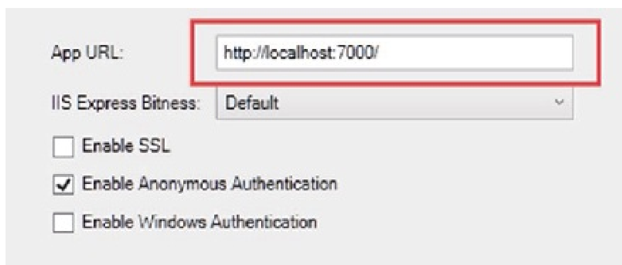


图20-1　设置应用程序URL

启动应用程序，填写表单，然后单击Add按钮；应用程序将为模型添加新的数据，如图20-2所示。但你对存储库所做的更改不会持久保留，在应用程序停止或重新启动时将会丢失。

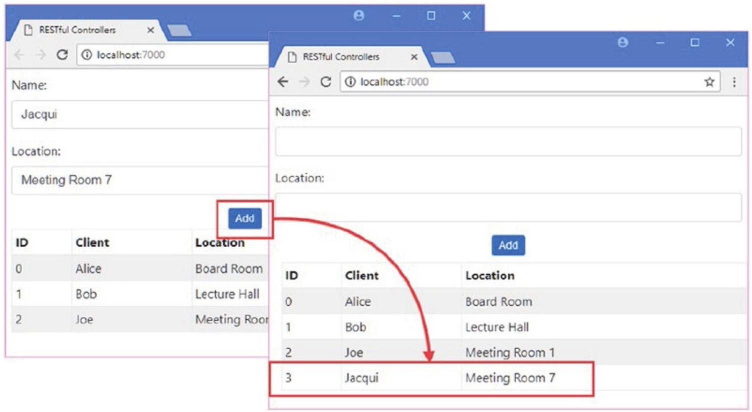


图20-2　启动示例应用程序

20.2　REST控制器的作用

这个示例应用程序是经典的Web应用程序。示例应用程序中的所有逻辑都存在于服务器上，包含在C#类中，这使得它们易于管理、测试和维护。但是，以这种方式设计的应用程序在速度、效率和开放性方面可能会有严重的缺陷。

20.2.1　速度问题

此时，示例应用程序是同步的Web应用程序。当用户单击Add按钮时，浏览器将POST请求发送到

服务器，等待响应，然后呈现接收到的HTML。在此期间，用户无法做任何事情，只能等待。当浏览器和服务器位于同一台计算机上时，等待时间在开发过程中是不可察觉的；然而，部署的应用程序受实际容量和延迟的限制，并且同步应用程序要求用户等待响应的时间可能很长。

同步应用程序不会总有速度问题。例如，如果你正在编写一个线路业务应用程序，用于单个位置，其中所有客户端都通过快速可靠的LAN连接，那么不会有需要解决的问题。如果你正在基础设施薄弱的地区为移动客户端编写应用程序，则速度问题可能很严重。

提　示

某些浏览器可让你模拟不同类型的网络，这可以作为有用的工具，用于查看用户是否可能接受将同步应用程序用于各种场景。例如，Google Chrome提供了名为网络限制的功能，可按F12键，在开发人员工具的Network部分找到。有一系列预定义的网络可用，也可以通过指定上传/下载速率和延迟来创建自己的网络。

20.2.2　效率问题

效率问题源于同步Web应用程序将浏览器视为仅用于显示服务器发送的HTML文档的HTML呈现引擎的方式。

当用户首先请求示例应用程序的默认URL时，

发回的HTML文档将包含浏览器需要显示应用程序内容的所有信息，包括以下信息：

- 内容依赖于Bootstrap CSS文件，如果缓存副本不可用，就应该下载该文件。
- 内容包含配置为向AddReservation操作发送POST请求的表单。
- 内容包含一个表格，这个表格包含3行。

示例应用程序很简单，初始请求导致服务器向客户端发送约1.3KB数据。但是，当用户提交表单时，客户端将再次重定向到Index操作，这将导致另外1.3KB数据反映单个表行的添加。浏览器已经呈现了表单和表格，但是这些表单和表格都被丢弃，并被替换为在很大程度上内容相同的全新表示形式。

你可能认为1.3KB数据并不多，但是，如果考虑到有用内容与重复内容的比例，你将看到发送到

浏览器的绝大多数数据被浪费了。示例应用程序是故意简化的；很少有应用程序需要这么少的HTML标记，并且随着应用程序复杂性的增加，重复内容的数量将显著增加。

20.2.3　开放性问题

传统Web应用程序提出的最终问题是设计是关闭的，这意味着模型中的数据只能通过Home控制器提供的操作进行访问。当需要在另一个应用程序中使用底层数据时，封闭的应用程序成为问题，特别是当应用程序由不同的团队甚至不同的组织开发时。开发人员经常认为，应用程序的价值在于为用户提供互动，主要原因在于这些是我们花时间思考和写作的部分。一旦建立了应用程序并具有活跃的用户群，应用程序包含的数据往往将变得重要。

20.3　REST和API控制器

API控制器是MVC控制器，负责提供对应用程序数据的访问，而不将它们封装在HTML中。这允许检索或修改模型中的数据，而不必使用常规控制器提供的操作，例如示例中的Home控制器。

从应用程序传递数据的常见方法是使用称为REST的表示状态传输模式。没有关于REST的详细规范，这导致许多不同的方法包含在REST风格之下。但是，这在客户端Web应用程序开发中是有用的。

Web服务的核心前提是采用HTTP的特征，以便请求方法（也称为谓词）指定服务器执行的操作，请求URL指定一个或多个数据对象，以应用操作。

作为示例，下面是一个可能在示例应用程序中引用的特别预留的URL：

```
/api/reservations/1
```

URL的第一部分——api用于将应用程序的数据部分与生成HTML的标准控制器分离。下一部分——reservations指示将要操作的对象集合。最后一部分——1指定对象集合中的单个对象。在示例应用程序中，这就是唯一标识对象并将在URL中使用的ReservationId属性值。

请识别对象的URL与HTTP方法组合以指定操作。表20-3列出了常用的HTTP方法以及与示例URL结合使用的方法，还列出了每个方法和URL组合的请求及响应中包含哪些数据（有效负载）的详细信息。处理这些请求的API控制器使用响应状态代码来报告请求的结果。

表**20-3**　常用的**HTTP**方法以及与示例**URL**结合使用的方法

请求方法	**URL**地址	描述	载荷

GET	/api/reservations	检索所有对象	包含预留对象的完整集合
GET	/api/reservations/1	检索ReservationId为1的Reservation对象	包含指定的Reservation对象
POST	/api/reservation	创建新的Reservation对象	包含创建Reservation对象所需的其他属性的值。响应中包含存储的对象，确保客户端接收保存的数据
PUT	/api/reservation	更新现有的Reservation对象	包含更改指定Reservation对象的属性所需的值。响应中包含存储的对象，确保客户端接收保存的数据
PATCH	/api/reservation/1	修改现有的Reservation对象，该对象的ReservationId为1	包含一组应该应用于指定的Reservation对象的修改，是对更改已应用的确认
DELETE	/api/reservation/1	删除ReservationId为1的Reservation对象	请求或响应中没有有效负载

遵循REST约定不是必需的，但这么做有助于使应用程序更容易使用，因为许多已建立的Web应用程序广泛采用了相同的方法。

20.3.1　创建API控制器

创建API控制器的过程基于创建标准控制器的方法，还有一些其他功能可以帮助指定提供给客户端的API。为了演示，添加一个名为ReservationController.cs的类文件到Controllers文件夹中，并用它定义代码清单20-10所示的类。稍后将介绍这个控制器提供的功能。

代码清单20-10　Controllers文件夹下的ReservationController.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;
using Microsoft.AspNetCore.JsonPatch;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ReservationController : Controller {
        private IRepository repository;

        public ReservationController(IRepository repo)
=> repository = repo;

        [HttpGet]
        public IEnumerable<Reservation> Get() => reposi
```

```
tory.Reservations;

        [HttpGet("{id}")]
        public Reservation Get(int id) => repository[id
];

        [HttpPost]
        public Reservation Post([FromBody] Reservation
res) =>
            repository.AddReservation(new Reservation {
                ClientName = res.ClientName,
                Location = res.Location
            });

        [HttpPut]
        public Reservation Put([FromBody] Reservation r
es) =>
            repository.UpdateReservation(res);

        [HttpPatch("{id}")]
        public StatusCodeResult Patch(int id,
                [FromBody]JsonPatchDocument<Reservation
> patch) {
            Reservation res = Get(id);
            if (res != null) {
                patch.ApplyTo(res);
                return Ok();
            }
            return NotFound();
        }

        [HttpDelete("{id}")]
        public void Delete(int id) => repository.Delete
Reservation(id);
    }
}
```

提　示

请记住，控制器类可以在项目中的任何位置进行定义，而不仅仅是在Controllers文件夹中。对于大型和复杂项目，与常规HTML控制器分开，将定义的API控制器完全放置在子文件夹甚至单独的文件夹中是有帮助的。

API控制器的工作方式与常规控制器相同，这意味着可以创建POCO控制器，或者从控制器基类派生类，这样可以更方便地访问请求上下文数据。

适应REST模式

REST模式在如何将Web应用程序API呈现给客户端方面，在一定程度上鼓励教条主义。REST不是标准的甚至不是定义明确的模式，但REST提供了一些有用的方法，可以让ASP.NET Core MVC应用程序更容易采用，但是有一种趋势会刺激那些对作为REST计数有固定观点的程序员。

在表20-3中，为POST和PUT操作列出的URL不唯一标识资源，有些人认为这是一个基本的REST特性。在POST操作的情况下，预留对象的唯一标识符由模型分配，这意味着客户端无法将其作为URL的一部分提供。在PUT操作的情况下，第26章描述的MVC模型绑定功能是应用代码清单20-10中的FromBody特性的原因，使得从请求体接收到要

修改的Reservation对象的详细信息更加容易。所以，这就是Reservation控制器期望找到ReservationId值的地方，该值用于标识要修改的模型对象。

与所有模式一样，REST 模式是起点，这不是必须不惜一切代价遵守的严格标准，唯一重要的是编写可以被理解、测试和维护的代码。通过适应MVC应用程序的性质和存储库的设计可造就更简单的应用程序，同时仍然为客户提供有用的API。建议将模式视为按自己需求所选的指导原则，这与REST模式一样，对于MVC本身也是如此。

---

1. 定义路由

到达API控制器的路由只能使用Route特性进行

定义，无法在Startup类的应用程序配置中定义。API控制器的惯例是使用前缀为api的路由，后跟控制器的名称，以便通过URL——/api/reservation访问如代码清单20-10所示的ReservationController控制器，如下所示：

```
...
[Route("api/[controller]")]
public class ReservationController : Controller {
...
```

2．声明依赖关系

API控制器以与常规控制器相同的方式实例化，这意味着它们可以声明将使用service provider解析的依赖关系。ReservationController类在IRepository接口上声明了构造函数依赖关系，它将被解析为提供对模型中数据的访问：

```
...
public ReservationController(IRepository repo) => repos
itory = repo;
```

```
...
```

3. 定义操作方法

每个操作方法都使用一个特性进行装饰，该特性指定了能够接受的HTTP方法，如下所示：

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Reservations;
...
```

HttpGet特性是用于将对操作方法的访问限制为具有特定HTTP方法或谓词的请求集合之一。HTTP特性参见表20-4。

表**20-4　HTTP**特性

HTTP特性	描述
HttpGet	指定只能由使用GET方法的HTTP请求调用
HttpPost	指定只能由使用POST方法的HTTP请求调用

HttpDelete	指定只能由使用DELETE方法的HTTP请求调用
HttpPut	指定只能由使用PUT方法的HTTP请求调用
HttpPatch	指定只能由使用PATCH方法的HTTP请求调用
HttpHead	指定只能由使用HEAD方法的HTTP请求调用

通过将路由片段作为HTTP方法特性的参数包括在内，可以进一步细化路由，如下所示：

```
...
[HttpGet("{id}")]
public Reservation Get(int id) => repository[id];
...
```

路由片段{id}可与应用于控制器的Route特性定义的路由以及基于HTTP方法的约束相结合。在这种情况下，这意味着可以通过发送一个URL匹配/api/reservations/{id}路由模式的GET请求来执行此操作，然后使用id片段来标识应该检索的Reservation对象。

请注意，为API控制器生成的路由不包含{action}片段变量，这意味着操作方法的名称不是目标特定方法所需URL的一部分。API控制器中的所有操作都通过相同的基本URL（例如/api/reservation）达成，HTTP方法和可选参数片段可用于区分它们。

4．定义操作结果

API控制器的操作方法不依赖于ViewResult对象来呈现结果，因此在传递数据时不需要视图。API控制器的操作方法直接返回数据对象，如下所示：

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Res
ervations;
...
```

以上操作返回一系列Reservation对象，并使

MVC负责将其序列化为可由客户端处理的格式。20.4节将更详细地解释这个过程。

自定义API结果

API控制器最有吸引力的是可以从操作方法返回C#对象，并让MVC弄清楚如何处理它们。MVC很擅长处理，例如，如果从API控制器的操作方法返回null，客户端将被发送204-No Content响应。

但API控制器也可以使用常规控制器的功能，这意味着可以通过从指定要发送何种结果的操作方法返回IActionResult来覆盖默认行为。作为示例，这里是来自示例控制器的操作方法的实现，示例控制器向与模型中的对象匹配失败的查询发送404-Not Found响应：

```
...
[HttpGet("{id}")]
public IActionResult Get(int id) {
    Reservation result = repository[id];
    if (result == null) {
        return NotFound();
    } else {
        return Ok(result);
    }
}
...
```

如果指定ID的存储库中没有对象，那么将调用NotFound方法，该方法将创建一个NotFoundResult对象，这反过来又导致发送到客户端的404-Not Found响应。如果存储库中有对象，就调用Ok方法来创建ObjectResult对象。Ok方法允许在返回IActionResult的操作中向客户端发送对象，如第17章所述。通常不需要覆盖默认的API控制器响应，但是如果需要，则可以使用全部action结果。

20.3.2　测试API控制器

这里有很多可用于帮助测试Web应用程序API的工具。好的例子包括Fiddler，它是一个独立的HTTP调试工具；还有Swashbuckle，它是一个NuGet软件包，可将摘要页面添加到描述其API操作并允许对它们在应用程序中进行测试。

最简单的测试API控制器的方法是使用PowerShell，这样可以轻松地从Windows命令行创建HTTP请求，并且可以让你专注于API操作的结果，而无须挖掘细节。PowerShell起源于Windows，但现在也可用于Linux系统和macOS。

下面的内容将告诉你如何使用PowerShell来测试由Reservation控制器提供的每个操作。可以打开一个新的PowerShell窗口来运行测试命令或使用PowerShell的Visual Studio包管理器控制台窗口。

1．测试**GET**操作

要测试Reservation API控制器提供的GET操作，请从Visual Studio的Debug菜单中选择Start Without Debugging以启动应用程序，然后等待，直到看到Home控制器提供的同步响应。应用程序运行后，打开PowerShell窗口并键入以下命令：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method GET
```

上述命令使用Invoke-RestMethod PowerShell cmdlet将GET请求发送到URL——/api/reservation。结果被解析和格式化，以使数据容易阅读，如下所示：

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
```

服务器使用模型中包含的Reservation对象的JSON表示形式来响应GET请求，Invoke-

RestMethod cmdlet以表格形式显示。

理解JSON

JSON（JavaScript对象表示法）已成为Web应用程序的标准数据格式。JSON之所以变得流行，是因为它简单且易于使用。由于JSON格式与JavaScript代码中的文字对象类似，因此JavaScript代码中的JSON数据处理尤其容易。现代浏览器提供生成和解析JSON数据的内置支持，流行的JavaScript库（如jQuery）会自动与JSON互相转换。虽然JSON从JavaScript演变而来，但它的结构很容易让C#开发人员阅读和理解。举例来说，以下是示例应用程序中API控制器的响应：

```
...
[{"reservationId":0,"clientName":"Alice","location":"Board Room"},
```

```
 {"reservationId":1,"clientName":"Bob","location":"Lect
ure Hall"},
 {"reservationId":2,"clientName":"Joe","location":"Meet
ing Room 1"}]
...
```

上述JSON字符串描述了一个对象数组。该数组由[和]字符表示，每个对象使用{和}字符表示。对象是键/值对的集合，用冒号分隔键和值，并且用逗号分隔键值对。这在广义上与MemoryRepository类中用于定义代码清单20-3所示数据的C#文字语法类似。

```
...
new List<Reservation> {
    new Reservation { ClientName = "Alice", Location =
"Board Room" },
    new Reservation { ClientName = "Bob", Location = "L
ecture Hall" },
    new Reservation { ClientName = "Joe", Location = "M
eeting Room 1" }
...
```

但请注意，MVC会将C#约定（ClientName，具有初始大写字母）的属性名中的初始大写字母更

改为JavaScript约定（clientName，带有初始小写字母）。

即使这些格式不完全相同，也仍有很多相似之处，C#开发人员读取和理解JSON数据并不费力。你不需要了解大多数Web应用程序的JSON详细信息，因为MVC做了很大提升，可以从www.json.org了解更多有关JSON的信息。

Reservation控制器提供两种GET操作。当把GET请求发送到/api/reservation时，会返回包含所有对象的响应。为了检索单个对象，可将这个对象的ReservationId值指定为URL中的最后一个片段，如下所示：

```
Invoke-RestMethod http://localhost:7000/api/reservation/1 -Method GET
```

以上命令请求ReservationId值为1的Reservation对象，并产生以下结果：

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Lecture Hall
```

2．测试**POST**操作

API控制器提供的所有操作都可以使用PowerShell进行测试，尽管命令的格式可能有点怪异。以下命令向API控制器发送POST请求，以在存储库中创建一个新的Reservation对象，并在响应中接收发回的数据：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method POST -Body
(@{clientName="Anne"; location="Meeting Room 4"} | Conv
ertTo-Json) -ContentType
"application/json"
```

以上命令使用-Body参数来指定请求的主体，请求的主体被编码为JSON。-ContentType参数用于

设置请求的Content-Type标头。以上命令将产生以下结果：

```
reservationId clientName location
------------- ---------- --------
            3 Anne       Meeting Room 4
```

POST操作使用clientName和location的值创建了一个Reservation对象，并将这个新对象的JSON表示返回给客户端，其中包括已分配给新对象的ReservationId值。这可能看起来像客户端只是接收到请求中发送给服务器的数据值，但这种方法可以确保客户端正在使用与服务器正在使用的相同的数据，并且可以满足服务器对从客户端接收的数据执行任何格式化或翻译操作的需求。要查看POST请求的效果，请将另一个GET请求发送到API /api/reservation，如下所示：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method GET
```

客户端返回的数据反映了新的Reservation对象已添加。

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

3．测试PUT操作

PUT操作用于更改模型中的现有对象。对象的ReservationId值被指定为请求URL的一部分，并且clientName和location的值在请求的主体中提供。以下PowerShell命令将发送一个PUT请求来修改一个Reservation对象：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method PUT -Body
(@{reservationId="1"; clientName="Bob"; location="Media
 Room"} | ConvertTo-Json)
-ContentType "application/json"
```

上述PUT请求会更改ReservationId值为1的Reservation对象，并为location属性指定新值。如果运行上述命令，你将看到以下响应，这表示更改已完成。

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Media Room
```

为了查看PUT请求的效果，需要向API /api/reservation发送另一个GET请求，如下所示：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method GET
```

客户端返回的数据反映了新的Reservation对象已添加。

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

4．测试**PATCH**操作

PATCH操作用于修改模型中的现有对象。许多应用程序使用PUT请求并完全忽略PATCH，如果客户端可以访问模型中对象定义的所有属性，那么这将是一种合理的方法。但是在复杂的应用程序中，出于安全原因，客户端可能会收到一组特定的属性值，这会阻止它们作为PUT请求的一部分发送完整的对象。PATCH请求更具选择性，允许客户端为对象指定一组精细更改。

ASP.NET Core MVC支持使用JSON Patch标准，从而允许以统一的方式指定更改。这里不打算详细介绍JSON Patch标准，可以从IETF网站查看。对于示例应用程序，客户端将发送HTTP PATCH请求中的数据，如下所示：

```
[
 { "op": "replace", "path": "clientName", "value": "Bob
"},
```

```
 { "op": "replace", "path": "location", "value": "Lecture Hall"}
]
```

JSON Patch文档将被表示为一系列操作。每个操作都有op属性，用于指定操作的类型。另外，每个操作还有path属性，用于指定操作的应用位置。

对于示例应用程序（事实上，对于大多数应用程序），仅需要替换操作，替换操作用于更改属性的值。为clientName和location属性设置新值，而要修改的对象由请求URL标识。ASP.NET Core MVC将自动处理JSON数据并将其作为JsonPatchDocument<T>对象呈现给操作方法，其中T是要修改的模型对象的类型。然后，可以使用JsonPatchDocument<T>对象的ApplyTo方法从存储库中修改对象。以下是发送PATCH请求的PowerShell命令：

```
Invoke-RestMethod http://localhost:7000/api/reservation
```

```
/2 -Method PATCH -Body (@
{ op="replace"; path="clientName"; value="Bob"},@{ op="
replace"; path="location";
value="Lecture Hall"} | ConvertTo-Json) -ContentType "a
pplication/json"
```

上述请求要求服务器修改ReservationId为2的Reservation对象的clientName和location属性。要查看PUT请求的效果，请向API /api/ reservation发送GET请求，如下所示：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method GET
```

客户端返回的数据反映了新的Reservation对象已添加。

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Bob        Lecture Hall
            3 Anne       Meeting Room 4
```

5．测试**DELETE**操作

发送一个DELETE请求，用于从存储库中删除一个Reservation对象，如下所示：

```
Invoke-RestMethod http://localhost:7000/api/reservation
/2 -Method DELETE
```

Reservation控制器中接收DELETE请求的操作不会返回结果，因此在命令完成后不会显示任何数据。要查看删除效果，请使用以下命令请求存储库中的内容：

```
Invoke-RestMethod http://localhost:7000/api/reservation
 -Method GET
```

ReservationId为2的Reservation对象已从存储库中删除。

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            3 Anne       Meeting Room 4
```

20.3.3 在浏览器中使用API控制器

通过定义API控制器虽然解决了应用程序的开放性问题，但是没有解决速度或效率问题。为此，这里需要更新应用程序的HTML部分，以便靠JavaScript来向API控制器发出HTTP请求以执行数据操作。

在浏览器中，异步HTTP请求通常称为Ajax请求，其中Ajax是Asynchronous JavaScript and XML的缩写。XML数据格式近年来已不那么受欢迎，但Ajax仍然用于引用异步HTTP请求，即使它们返回JSON数据。更广泛地说，本节描述的技术是单页应用程序的基础，其中HTML单页应用程序中的JavaScript用于为应用程序的多个部分提取数据，以生成动态显示的内容。

注　意

客户端开发是本书范围内的讨论主题。这里只创建了基本的异步HTTP请求，而不进行详细解释。要想了解更多的信息，请参阅由Apress出版的*Pro ASP.NET Core MVC Client Development*一书，其中详细介绍了如何使用JavaScript和jQuery来创建从API控制器提供服务的单页应用程序。

浏览器提供了用于创建Ajax请求的JavaScript API，但处理方式有点怪异，不同的浏览器在实现一些可选功能的方式上有一些差异。创建Ajax请求的最简单方法是使用jQuery库，jQuery是客户端开发中非常有用的工具。代码清单20-11将jQuery包添

加到了bower.json文件中。

代码清单20-11　在ApiControllers文件夹下的bower.json文件中添加jQuery包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

实际上，由于某些Bootstrap功能取决于jQuery，Bower将会在wwwroot/lib文件夹中安装jQuery包。为了使用jQuery提供的功能，这里创建wwwroot/js文件夹，并添加一个名为client.js的JavaScript文件，内容如代码清单20-12所示。

代码清单20-12　wwwroot/js文件夹下的client.js文件的内容

```
$(document).ready(function () {

    $("form").submit(function (e) {
```

```
        e.preventDefault();
        $.ajax({
            url: "api/reservation",
            contentType: "application/json",
            method: "POST",
            data: JSON.stringify({
                clientName: this.elements["ClientName"]
.value,
                location: this.elements["Location"].val
ue
            }),
            success: function(data) {
                addTableRow(data);
            }
        })
    });
});

var addTableRow = function (reservation) {
    $("table tbody").append("<tr><td>" + reservation.re
servationId + "</td><td>"
        + reservation.clientName + "</td><td>"
        + reservation.location + "</td></tr>");
}
```

当用户在浏览器中提交表单，将表单数据编码为JSON，并使用HTTP POST请求将它们发送到服务器时，client.js文件中的JavaScript文件将会响应。服务器返回的JSON数据将自动由jQuery解析，然后用于向HTML表格中添加一行。代码清单20-13

更新了布局以包含jQuery库和client.js文件。

代码清单20-13　在_Layout.cshtml文件中添加JavaScript引用

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>RESTful Controllers</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
    <script src="lib/jquery/dist/jquery.js"></script>
    <script src="js/client.js"></script>
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

第一个script元素告诉浏览器加载jQuery库，第二个script元素指定包含自定义代码的文件。如果运行应用程序，你会发现这和使用HTML表单在应用程序的存储库中创建Reservation没有什么区别。但是，如果检查浏览器发送的HTTP请求，你会看到需要的数据少于同步版本的应用程序。在这里的测

试中，异步请求需要480字节的数据，大约是同步请求所需的40%。数据的大小倾向于比用于显示数据的HTML文档小得多，在这样的实际应用中，改进更为显著。

20.4　内容格式

当操作方法返回C#对象作为结果时，MVC必须弄清楚应该使用哪种数据格式对对象进行编码并发送给客户端。本节将解释什么是默认进程，以及默认进程如何受到客户端发送的请求和应用程序的配置的影响。为了帮助理解工作原理，这里将一个名为ContentController.cs的类文件添加到Controllers文件夹中，并用来定义代码清单20-14所示的API控制器。

代码清单20-14　Controllers文件夹下的ContentController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string
response";

        [HttpGet("object")]
        public Reservation GetObject() => new Reservati
on {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

这里指定静态的片段变量作为HttpGet特性的参数，用于控制器中的两个操作，这意味着它们可以通过请求URL /api/controller/string和/api/controller/object来访问。Content控制器不会松散地遵循REST模式，但可以使你能够轻松了解内容协商的工作原理。

MVC选择的内容格式取决于4个因素——客户端接收的格式，MVC可以生成的格式，操作指定的内容策略以及操作方法返回的类型。了解清楚这一切如何融合在一起是很复杂的，但好消息是，默认策略对大多数应用程序适用，你只需要了解当需要进行更改或未获得更改时幕后发生的情况，即可得到期望的格式。

20.4.1　默认内容策略

首先采用标准应用程序配置，当客户端和操作方法都不对可以使用的数据格式应用任何限制时，返回的结果是简单和可预测的。

- 如果操作方法返回一个字符串，那么不对这个字符串做修改并发送给客户端，将响应的Content-Type标头设置为text/plain。
- 对于所有其他数据类型，包括其他简单类型（如int），数据格式为JSON，并且将响应的

Content-Type标头设置为application/json。

字符串得到特殊处理的原因是它们在编码为JSON时会导致问题。当编码其他简单类型（例如C#中的int类型，值为2）时，结果是引用的字符串，比如"2"。当编码字符串时，会得到两组引号，所以"Hello"变成""Hello""。并不是所有客户端都能应对这种双重编码，所以使用text/plain格式更可靠，可以全面避免这个问题。这基本不是问题，因为极少有应用程序直接发送字符串值；以JSON格式发送对象更常见。可以通过使用PowerShell查看这两个结果。以下命令将调用GetString方法并返回一个字符串：

```
Invoke-WebRequest http://localhost:7000/api/content/string | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

以上命令将向URL /api/content/string发送GET

请求，并处理响应以显示Content-Type标头和响应的内容。

提　示

如果尚未执行Internet Explorer的初始设置，那么在使用Invoke-WebRequest cmdlet时可能会收到错误提示。这极可能发生在Edge已经替换的Windows 10计算机上。可以通过运行IE并选择所需的初始配置来修复此问题。

以上命令产生的输出结果如下：

```
Content-Type                    Content
```

```
------------               -------
text/plain; charset=utf-8 This is a string response
```

同样的命令也可以通过改变请求的URL来显示JSON格式，如下所示：

```
Invoke-WebRequest http://localhost:7000/api/content/obj
ect | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Co
ntent
```

以上命令产生的输出格式清晰，表明响应已编码为JSON：

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"reservationId":100,
                                  "clientName":"Joe",
                                  "location":"Board Room
"}
```

20.4.2　内容协商

大多数客户端将在请求中包含Accept标头，用于指定它们希望在响应中接收的数据格式，以一组

MIME类型表示。以下是Google Chrome在请求中发送的Accept标头：

```
Accept: text/html,application/xhtml+xml,application/xml
;q=0.9,image/webp,*/*;q=0.8
```

以上Accept标头表示Google Chrome可以处理HTML、XHTML格式（XHTML是XML兼容的HTML格式）、XML和WEBP图像格式。标头中的q值指定了有关偏好，默认值为1.0。为application/xml指定0.9的q值会告诉服务器Google Chrome可以接收XML数据，但Google Chrome通常会处理HTML或XHTML数据。最后一项/会告诉服务器Google Chrome可以接收任何数据格式，但q值表明这是所有指定类型的最低优先级。

（1）Google Chrome通常会接收HTML/XHTML数据或WEBP图像。

（2）如果这些格式不可用，下一种优选的格式是XML。

（3）如果以上格式都不可用，Google Chrome将接收任何格式的数据。

由此假设可以通过设置Accept标头来更改请求从MVC应用程序接收数据的格式，但不能以这种方式工作，或者更确切地说，现在还不能这样工作，因为需要做一些准备工作。首先需要一个PowerShell命令，使用Accept标头向GetObject方法发送GET请求，Accept标头指定客户端将仅接收XML数据。

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers @{Accept="application/
xml"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

以下是请求返回的结果，服务器返回了

application/json响应：

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"reservationId":100,
                                  "clientName":"Joe",
                                  "location":"Board Room
"}
```

包括Accept标头的请求对返回数据的格式没有影响，服务器已经向客户端发送了未指定格式的数据。因为在默认情况下MVC仅支持JSON，所以没有其他可以使用的格式。MVC发送JSON数据是希望客户端可以处理数据而不是返回错误信息，即使JSON格式不是请求的Accept标头指定的格式之一。

配置JSON Serializer

ASP.NET Core MVC使用流行的第三方JSON包Json.Net将对象序列化为JSON。默认配置适用于大

多数项目，但如果需要以特定方式创建JSON，则可以对它们进行更改。可以在Startup类中使用AddMvc().AddJsonOptions扩展方法，以提供对MvcJsonOptions对象的访问，进而配置Json.Net包。有关可用配置选项的详细信息，请参见newtonsoft网站。

启用**XML**格式

为了在工作中看到内容协商，必须给MVC一些用于对响应数据进行编码的格式选择。虽然JSON已经成为Web应用程序的默认格式，但MVC也可以支持XML作为编码数据，如代码清单20-15所示。

提　示

从

Microsoft.AspNetCore.Mvc.Formatters.OutputFormatte
类可以派生和创建自己的内容格式，但很少有人这么做，因为创建自定义数据格式不是在应用程序中展示数据的有效方式，而且最常见的格式JSON和XML已经实现了。

代码清单20-15　在ApiControllers文件夹下的Startup.cs文件中启用XML格式

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc().AddXmlDataContractSeriali
zerFormatters();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

当MVC只有JSON格式可用时，别无选择，只能将响应编码为JSON。现在有了另一选择，可以看到内容协商过程更加完善。

提　示

这里使用了代码清单20-15中的AddXmlDataContractSerializerFormatter扩展方法，但是你也可以使用AddXmlSerializerFormatters扩展方法，以提供对旧的序列化类的访问。如果需要为较早版本的.NET客户端生成XML内容，这可能会很有帮助。

以下是再次请求XML数据的PowerShell命令：

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers @{Accept="application/
xml"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

运行上述命令，你将看到服务器返回XML数据

而不是JSON，如下所示（为了简洁起见，这里省略了XML命名空间特性）：

```
Content-Type                Content
------------                -------
application/xml; charset=utf-8 <Reservation>
                                   <ClientName>Joe</Cli
entName>
                                   <Location>Board Room
</Location>
                                   <ReservationId>100</
ReservationId>
                               </Reservation>
```

20.4.3 指定action数据格式

你可以覆盖内容协商系统，并通过使用Produces特性直接在操作方法上指定数据格式，如代码清单20-16所示。

代码清单20-16 在Controllers文件夹下的ContentController.cs文件中指定数据格式

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;
```

```
namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string
response";

        [HttpGet("object")]
        [Produces("application/json")]
        public Reservation GetObject() => new Reservati
on {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

Produces特性是过滤器，可以更改ObjectResult对象的内容类型，MVC使用这些对象来表示API控制器中的action结果。Produces特性的参数不仅可以指定用于action结果的格式，还可以指定其他的类型。Produces特性将强制响应使用的格式，你可以通过运行以下PowerShell命令看到。

```
(Invoke-WebRequest http://localhost:7000/api/content/ob
```

```
ject -Headers
@{Accept="application/xml"}).Headers."Content-Type"
```

以上命令将显示GET请求URL /api/content/object返回的响应中的Content-Type标头的值。运行该命令后你会发现，请求的Accept标头指定应使用XML，返回的数据使用Produces特性指定的JSON格式。

20.4.4 从路由或查询字符串获取数据格式

Accept标头并不总是在编写客户端的程序员的控制之下，特别是在使用旧的浏览器或工具包进行开发的情况下。对于这种情况，允许通过用于定位操作方法或请求URL的查询字符串部分的路由来请求响应的数据格式是有帮助的。第一步是在Startup类中定义可用于引用路由或查询字符串格式的速记值。默认情况下有一个映射，其中json用作application/json的缩写。代码清单20-17添加了一个

额外的XML映射。

代码清单20-17　在ApiControllers文件夹下的Startup.cs文件中添加格式

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc()
                .AddXmlDataContractSerializerFormatters
()
                .AddMvcOptions(opts => {
                    opts.FormatterMappings.SetMediaType
MappingForFormat("xml",
                        new MediaTypeHeaderValue("appli
cation/xml"));
                });
```

```
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

MvcOptions.FormatterMappings属性用于设置和管理映射。在以上代码中，使用SetMediaTypeMapping- ForFormat方法创建了一个新的映射，以便简写的xml代表application/xml格式。下一步是将FormatFilter属性应用于操作方法，并可选地调整操作的路由，使它包含format变量，如代码清单20-18所示。

代码清单20-18　在Controllers文件夹下的ContentController.cs文件中应用FormatFilter特性

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

```

```
namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string
response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml"
)]
        public Reservation GetObject() => new Reservati
on {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

这里已经将FormatFilter特性应用于GetObject方法，并修改了操作的路由，使之包含可选的format字段。不必将Produces特性与FormatFilter特性一起使用，但如果这样做，那么只有指定Produces特性配置格式的请求才能正常工作。指定Produces特性尚未配置的格式的请求将收到404-Not Found响应。

如果不应用Produces特性，那么请求可以使用MVC被配置的任何格式。

这里还将application/xml格式添加到Produces特性中，以使操作方法支持对JSON和XML的请求。

以下PowerShell命令将xml格式指定为请求URL的一部分：

```
(Invoke-WebRequest http://localhost:7000/api/content/object/xml).Headers."Content-Type"
```

运行以上命令显示响应的内容类型，如下所示：

```
application/xml; charset=utf-8
```

使用FormatFilter特性查找名为format的路由片段变量，获取其中包含的速记值，并从应用程序配置中检索关联的数据格式，然后用于响应。如果没

有可用的路由数据，就检查查询字符串。以下是使用查询字符串请求XML的PowerShell命令：

```
(Invoke-WebRequest http://localhost:7000/api/content/object?format=xml).Headers."Content-Type"
```

FormatFilter特性找到的格式将覆盖Accept标头指定的任何格式，即使在使用不允许Accept标头设置的工具包和浏览器的情况下，也可以将格式选择权掌握在客户端开发人员手中。

20.4.5 启用完成内容协商

对于大多数应用程序来说，当没有其他格式可用时发送JSON数据是明智的，因为Web应用程序的客户端很可能错误设置了接收标头，而不是不能处理JSON。也就是说，如果不管Accept标头是什么JSON都被返回，一些应用程序将不得不处理那些导致问题的客户端。获取内容协商需要在Startup类

中更改两个配置，如代码清单20-19所示。

代码清单20-19　在ApiControllers文件夹下的Startup.cs文件中启用完成内容协商

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc()
                .AddXmlDataContractSerializerFormatters
()
                .AddMvcOptions(opts => {
                    opts.FormatterMappings.SetMediaType
MappingForFormat("xml",
                        new MediaTypeHeaderValue("appli
cation/xml"));
                    opts.RespectBrowserAcceptHeader = t
```

```
rue;
                    opts.ReturnHttpNotAcceptable = true;

                });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

RespectBrowserAcceptHeader选项用于控制是否完全遵守Accept标头。如果没有合适的格式可用，ReturnHttpNotAcceptable选项用于控制是否将406-Not Acceptable响应发送到客户端。

这里还必须从操作方法中删除Produces特性，以使内容协商过程不被覆盖，如代码清单20-20所示。

代码清单20-20　在Controllers文件夹下的ContentController.cs

文件中删除Produces特性

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string
response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        //[Produces("application/json", "application/xm
l")]
        public Reservation GetObject() => new Reservati
on {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

这里有一个PowerShell命令，用于向/api/content/object发送GET请求，并使用Accept标头指定应用程序无法提供的内容类型：

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/custom"}
```

如果运行以上命令，你将看到406错误消息，指示服务器无法提供请求的格式。

20.4.6 接收不同的数据格式

当客户端向控制器发送数据（例如POST请求）时，可以使用Consumes特性指定不同的操作方法来处理特定的数据格式，如代码清单20-21所示。

代码清单20-21 在Controllers文件夹下的ContentController.cs文件中处理不同的数据格式

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {
    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
```

```
        public string GetString() => "This is a string 
response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        //[Produces("application/json", "application/xm
l")]
        public Reservation GetObject() => new Reservati
on {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };

        [HttpPost]
        [Consumes("application/json")]
        public Reservation ReceiveJson([FromBody] Reser
vation reservation) {
            reservation.ClientName = "Json";
            return reservation;
        }

        [HttpPost]
        [Consumes("application/xml")]
        public Reservation ReceiveXml([FromBody] Reserv
ation reservation) {
            reservation.ClientName = "Xml";
            return reservation;
        }
    }
}
```

ReceiveJson和ReceiveXml操作都接收POST请

求，它们之间的区别在于使用Consumes特性指定的数据格式，检查Content-Type标头，以确定操作方法是否可以处理请求。当有请求的Content-Type设置为application/json时，使用ReceiveJson方法；当ContentType设置为application/xml时，使用ReceiveXml方法。

20.5　小结

本章介绍了API控制器在MVC应用程序中的作用，演示了如何创建和测试API控制器，还简要演示了如何使用jQuery进行异步HTTP请求，并解释了内容的格式化过程。下一章将更详细地解释视图和视图引擎的工作原理。

第21章　视图

第17章讲述了操作方法如何返回ViewResult对象，该对象用于告诉MVC渲染视图并向客户端返回HTML响应。

在本书中，你已经看到许多示例中使用的视图，大致了解了它们的作用，本章将对视图进行更深入的介绍。

本章首先展示MVC如何使用视图引擎处理ViewResult对象，包括演示如何创建自定义视图引擎，然后介绍如何有效地使用内置的Razor视图引擎，包括使用分部视图和布局部分，这是涉及有效进行MVC开发的重要内容。表21-1介绍了视图的背景。

表**21-1**　视图的背景

问题	答案
什么是视图?	视图是用于向用户显示内容的MVC模式的一部分。在ASP.NET Core MVC应用程序中，视图是包含HTML元素和C#代码的文件，它们将被处理以生成响应
视图有什么作用?	视图允许将数据的呈现与处理请求的逻辑分开。视图还允许在整个应用程序中应用相同的演示文稿，因为许多控制器可以使用相同的视图
如何使用视图?	大多数MVC应用程序使用Razor视图引擎，这可以轻松地混合HTML和C#内容。如第17章所述，视图可通过返回ViewResult对象作为操作方法的结果得以选择
在使用过程中是否有什么陷阱或局限性?	你可能需要一段时间才能习惯使用Razor。本章将解释Razor如何工作，这有助于揭示一些常用操作
是否有其他的替代方案?	有一些第三方视图引擎可用于MVC，但是它们的使用很有限

表21-2列出了本章要介绍的操作。

表**21-2**　本章要介绍的操作

操作	方法	代码清单

创建自定义视图引擎	实现IViewEngine和IView接口	代码清单21-3～代码清单21-6
轻松创建混合了HTML和C#代码的响应	使用Razor视图引擎	代码清单21-7～代码清单21-11
定义用于布局的内容区域	使用Razor部分	代码清单21-12～代码清单21-18
创建可重用的标记片段	使用分部视图	代码清单21-19～代码清单21-22
在视图中添加JSON内容	使用@ Json.Serialze表达式	代码清单21-23～代码清单21-25
更改Razor搜索视图的位置	创建视图位置扩展	代码清单21-26～代码清单21-30

21.1　准备示例项目

在本章中，使用ASP.NET Core Web Application（.NET Core）模板创建一个名为Views的Empty项目。为了启用MVC框架和其他对开发有用的中间件，对Startup类进行代码清单21-1所示的

更改。

代码清单21-1　Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

创建Controllers文件夹，添加一个名为

HomeController.cs的类文件，并用它定义控制器，如代码清单21-2所示。

代码清单21-2　Controllers文件夹下的HomeController.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Views.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello, World";
            ViewBag.Time = DateTime.Now.ToString("HH:mm
:ss");
            return View("DebugData");
        }

        public ViewResult List() => View();
    }
}
```

21.2　创建自定义视图引擎

本节将深入剖析如何创建自定义视图引擎。在

绝大多数项目中并不需要执行此操作，因为MVC提供了Razor视图引擎，第5章介绍了Razor的用法，Razor将出现在本书的所有示例中。

创建自定义视图引擎的价值在于查看后台发生的情况，并扩展你对MVC运行方式的了解，包括了解视图引擎在将ViewResult转换为对客户端的响应时有多少自由度。

视图引擎是实现了IViewEngine接口的类，定义在Microsoft.AspNetCore.Mvc.ViewEngines命名空间中。以下是IViewEngine接口的定义：

```
namespace Microsoft.AspNetCore.Mvc.ViewEngines {
    public interface IViewEngine {

        ViewEngineResult GetView(string executingFilePath, string viewPath,
            bool isMainPage);

        ViewEngineResult FindView(ActionContext context, string viewName,
            bool isMainPage);
    }
```

```
}
```

视图引擎的作用是将对视图的请求转换为ViewEngineResult对象。当MVC需要视图时，将调用GetView方法，使视图引擎根据视图名称来提供视图。

视图引擎所做的工作是为MVC提供可用于生成响应的ViewEngineResult对象。ViewEngineResult类不能直接实例化，而是提供用于创建实例的静态方法，如表21-3所示。

表**21-3　ViewEngineResult**类的静态方法

静态方法	描述
Found(name,view)	为MVC提供请求的视图，该视图是使用view参数设置的。视图实现了IView接口
NotFound(name,locations)	创建一个ViewEngineResult对象，以告诉MVC找不到请求的视图。locations参数是描述视图引擎查看视图位置的字符串值的枚举

在编写视图引擎时，可以选择表21-3中描述的方法之一来指示视图请求的结果。Found方法会创建一个指示请求成功的ViewEngineResult对象，并为MVC提供要处理的视图。NotFound方法会创建一个ViewEngineResult对象，指示不成功的请求，并向MVC提供视图引擎在查找视图时搜索过的位置清单（还将作为错误消息的一部分显示给开发人员）。

视图引擎系统的另一个构建模块是IView接口，用于描述视图提供的功能，而不考虑创建它们的视图引擎。以下为IView接口的定义：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.ViewEngines {

    public interface IView {

        string Path { get; }
        Task RenderAsync(ViewContext context);
    }
```

```
}
```

Path属性返回视图的路径，它假定视图为定义在磁盘上的文件。RenderAsync方法由MVC调用以生成对客户端的响应。通过从ActionContext派生的ViewContext类的实例将Context数据提供给视图。除从父类继承的Context属性（用于提供对请求的访问、路由数据、控制器等）之外，ViewContext类还提供了渲染响应中有用的属性，见表21-4。

表**21-4　ViewContext**有用的属性

属性	描述
ViewData	返回一个ViewDataDictionary对象，该对象包含控制器提供的视图数据
TempData	返回包含临时数据的字典
Writer	返回一个TextWriter对象，用于写入视图的输出

这些属性中最有用的是ViewData，它返回一个

ViewDataDictionary对象。ViewDataDictionary类定义了一些有用的属性，可以访问视图模型、视图包和视图模型元数据。表21-5列出了这些有用的属性。

表**21-5　ViewDataDictionary**有用的属性

属性	描述
Model	返回由控制器提供的模型数据
ModelMetadata	返回一个ModelMetadata对象，该对象可用于反映模型数据的类型
ModelState	返回模型的状态
Keys	返回可用于访问ViewBag数据的键值枚举

查看IviewEngine、ViewEngineResult和IView如何结合在一起工作的最简单方法是创建视图引擎。这里将创建一个简单的视图引擎，返回一种视图，以呈现包含有关请求的信息以及由操作方法生成的

视图数据的结果。这种方法有助于演示视图引擎的运行方式，而不会影响解析视图模板和重新创建Razor提供的其他功能。

21.2.1　创建自定义IView

本节将从创建IView接口的实现开始。在示例项目中创建Infrastructure文件夹，并添加一个名为DebugDataView.cs的类文件，如代码清单21-3所示。

代码清单21-3　Infrastructure文件夹下的DebugDataView.cs文件的内容

```
using System;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewEngines;

namespace Views.Infrastructure {

    public class DebugDataView : IView {
        public string Path => String.Empty;
```

```
        public async Task RenderAsync(ViewContext conte
xt) {
            context.HttpContext.Response.ContentType =
"text/plain";

            StringBuilder sb = new StringBuilder();

            sb.AppendLine("---Routing Data---");
            foreach (var kvp in context.RouteData.Value
s) {
                sb.AppendLine($"Key: {kvp.Key}, Value:
{kvp.Value}");
            }
            sb.AppendLine("---View Data---");
            foreach (var kvp in context.ViewData) {
                sb.AppendLine($"Key: {kvp.Key}, Value:
{kvp.Value}");
            }

            await context.Writer.WriteAsync(sb.ToString
());
        }
    }
}
```

当渲染这个视图时，使用ViewContext参数向RenderAsync方法写入路由数据和视图数据的详细信息。响应是简单的文本，以上代码已经使用context对象设置响应的Content-Type标头为text/plain。不这么做的话，ASP.NET默认会使用

text/html，这将导致浏览器将数据显示为单个不间断的字符行。

21.2.2 创建IViewEngine实现

视图引擎的目的是生成一个ViewEngineResult对象，该对象包含一个IView或者一个用于搜索合适视图的位置清单。既然有了IView实现，就可以创建视图引擎。在Infrastructure文件夹中添加一个名为DebugDataViewEngine.cs的类文件，内容如代码清单21-4所示。

代码清单21-4 Infrastructure文件夹下的DebugDataViewEngine.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewEngines;

namespace Views.Infrastructure {

    public class DebugDataViewEngine : IViewEngine {

        public ViewEngineResult GetView(string executin
```

```
gFilePath, string viewPath,
                bool isMainPage) {
            return ViewEngineResult.NotFound(viewPath,
                new string[] { "(Debug View Engine - Ge
tView)" });
        }

        public ViewEngineResult FindView(ActionContext
context, string viewName,
                bool isMainPage) {
            if (viewName == "DebugData") {
                return ViewEngineResult.Found(viewName,
 new DebugDataView());
            } else {
                return ViewEngineResult.NotFound(viewNa
me,
                    new string[] { "(Debug View Engine
- FindView)" });
            }
        }
    }
}
```

这个视图引擎中的GetView方法始终返回NotFound响应。FindView方法仅支持单个视图，名为DebugData。当接收到具有这一名称的视图请求时，就返回一个新的DebugDataView实例，如下所示：

```
...
if (viewName == "DebugData") {
    return ViewEngineResult.Found(viewName, new DebugDa
taView());
}
...
```

如果正在实现一个更完整的视图引擎，那么可以利用这个机会来搜索模板。但在这个示例中，只需要一个新的DebugDataView实例。如果收到除DebugData之外的视图请求，将创建NotFound响应，如下所示：

```
...
return ViewEngineResult.NotFound(viewName,
    new string[] { "(Debug View Engine - FindView)" });
...
```

ViewEngineResult.NotFound方法假定视图引擎具有查找视图所需的位置。这是一个合理的假设，因为视图通常是作为项目中的文件存储的模板文件。在这种情况下，没有任何地方可以查找视图，所以只返回一个虚拟位置，以指示调用哪个方法来

定位视图。

21.2.3　注册自定义视图引擎

视图引擎可通过配置MvcViewOptions对象在Startup类中注册，如代码清单21-5所示。

代码清单21-5　在Startup.cs文件中注册自定义视图引擎

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
            services.Configure<MvcViewOptions>(options
=> {
                options.ViewEngines.Insert(0, new Debug
DataViewEngine());
            });
```

```
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

MvcViewOptions类定义了ViewEngines属性，其值是一组IViewEngine对象。Razor可通过AddMvc方法添加到ViewEngine集合中，这里使用自定义类补充了默认视图引擎。

当MVC从一个操作方法接收到一个ViewResult对象时，它会调用MvcViewOptions.ViewEngines集合中包含的每个视图引擎的FindView方法，直到收到使用Found方法创建的ViewEngineResult对象。

如果两个或多个视图引擎能够为视图名称相同的请求提供服务，那么将视图引擎添加到

ViewEngines.Engines集合的顺序非常重要。要使你的视图优先，应将其插入视图引擎集合的开头，如代码清单21-5所示。

21.2.4　测试视图引擎

当应用程序启动时，浏览器将自动导航到项目的根URL，根URL将映射到Home控制器中的Index操作方法。这个操作方法使用View方法返回用于指定DebugData视图的ViewResult对象。

MVC将转向视图引擎的集合，并开始调用它们的FindView方法。由于请求的视图是自定义视图引擎设置为将要处理的视图，因此为MVC提供一个视图，该视图将产生图21-1所示的结果。

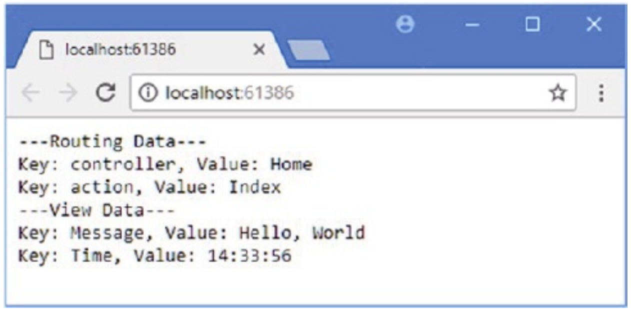


图21-1　使用自定义视图引擎

要查看当没有视图引擎可以提供视图时会发生什么，可请求URL——/Home/List，这将创建一个ViewResult对象，指向一个名为List的视图，List视图既不是Razor提供的，也不是自定义视图引擎提供的。你将看到图21-2所示的错误。

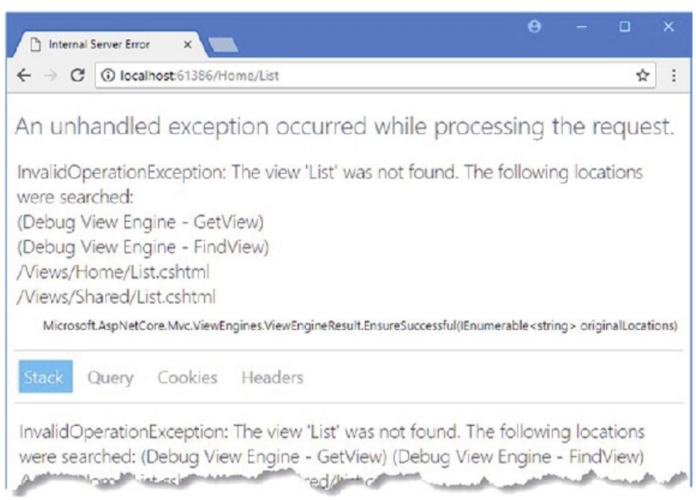


图21-2　请求无法提供的视图

可以看到，自定义视图引擎报告了寻找List视图的位置列表，同时还报告了Razor已检查的位置。

如果想确保只有你的视图引擎被使用，那就必须对视图引擎的集合调用Clear方法来删除Razor，如代码清单21-6所示。

代码清单21-6　在Views项目的Startup.cs文件中删除其他视图引擎

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
```

```
            services.Configure<MvcViewOptions>(options
=> {
                options.ViewEngines.Clear();
                options.ViewEngines.Insert(0, new Debug
DataViewEngine());
            });
        }
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

如果启动应用程序并再次导航到/Home/List，将只会使用自定义视图引擎，如图21-3所示。

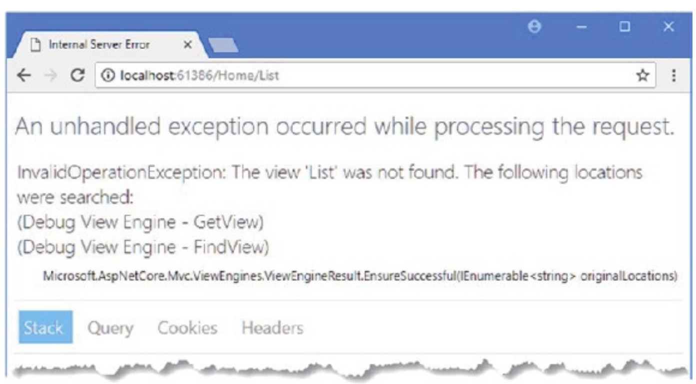


图21-3　在示例应用程序中仅使用自定义视图引擎

21.3　使用Razor引擎

可以通过仅实现两个接口来创建自定义视图引擎，MVC让添加或替换核心功能变得十分轻松。

视图引擎的复杂性来自视图模板系统，包括代码片段、支持布局和性能优化。这里在简单的自定义视图引擎中没有做以上这些事情，也没有太多需要，因为内置的Razor引擎提供了所有这些功能。实际上，几乎所有MVC应用程序所需的功能都可以在Razor中使用。只有极少量的项目需要自定义视图引擎。

第5章给出了关于Razor语法的介绍，本节将展示如何使用其他功能来创建和渲染Razor视图，还将讲述如何自定义Razor引擎。

21.3.1　准备示例项目

为了使用Razor，需要对示例应用做一些修改。首先，更改Home控制器的Index操作方法，以便选择默认视图并提供一些模型数据，如代码清单21-7所示。

代码清单21-7　修改Controllers文件夹下的HomeController.cs文件中的Index操作方法

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Views.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() =>
            View(new string[] { "Apple", "Orange", "Pea
r" });

        public ViewResult List() => View();
    }
}
```

为了向Index操作方法提供视图，创建Views/Home文件夹，并添加一个名为index.cshtml

的视图文件，内容如代码清单21-8所示。

代码清单21-8　Views/Home文件夹下的index.cshtml文件的内容

```
@model string[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    This is a list of fruit names:
    @foreach (string name in Model) {
        <span><b>@name</b></span>
    }
</body>
</html>
```

Index视图依赖于Bootstrap CSS库，要将Bootstrap添加到示例项目中，可使用Bower Configuration File模板在项目的根文件夹中创建

bower.json文件，内容如代码清单21-9所示。

代码清单21-9　bower.json文件的内容

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

在Views文件夹中添加一个名为_ViewImports.cshtml的视图文件，以启用内置的标签助手，如代码清单21-10所示。

代码清单21-10　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

最后的准备步骤是重新启动Startup类中的视图引擎以删除自定义引擎，并删除用来禁用Razor的Clear方法调用，如代码清单21-11所示。

代码清单21-11　重启Startup.cs文件中的视图引擎

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
            //services.Configure<MvcViewOptions>(option
s => {
            //    options.ViewEngines.Clear();
            //    options.ViewEngines.Insert(0, new Deb
ugDataViewEngine());
            //});
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
```

```
}
```

如果运行示例项目，你将看到图21-4所示的结果。

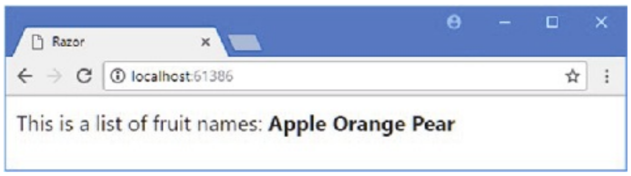


图21-4　运行示例项目

21.3.2　Razor视图

了解Razor的工作原理可以帮助将大量功能引入上下文中，并揭开CSHTML文件处理方式的谜团。

那么，Razor如何将HTML元素和C#语句相结合，并产生HTTP响应的内容？答案简单明了，它们建立在MVC功能的基础之上，你已经在前面的章节中了解了这些功能。Razor将CSHTML文件转换

为C#类并编译，然后在每次需要视图生成结果时创建新的实例。下面是Razor为Index.cshtml创建的C#类：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Razor;
using Microsoft.AspNetCore.Mvc.Razor.Internal;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Asp {

    public class ASPV_Views_Home_Index_cshtml : RazorPa
ge<string[]> {

        public IUrlHelper Url { get; private set; }

        public IViewComponentHelper Component { get; pr
ivate set; }

        public IJsonHelper Json { get; private set; }

        public IHtmlHelper<string[]> Html { get; privat
e set; }

        public override async Task ExecuteAsync() {
            Layout = null;

            WriteLiteral(@"<!DOCTYPE html><html><head>
                <meta name=""viewport"" content=""width
=device-width"" />
                <title>Razor</title>
```

```
                <link asp-href-include=""lib/bootstrap/
dist/css/*.min.css""
                    rel=""stylesheet"" />
                </head><body class=""m-1 p-1"">This is
a list of fruit names:");
            foreach (string name in Model) {
                WriteLiteral("<span><b>");
                Write(name);
                WriteLiteral("</b></span>");
            }
            WriteLiteral("</body></html>");
        }
    }
}
```

这里对类中的代码进行了整理，以便于阅读，并删除了Razor在生成类时为检测而添加的一些C#语句。后面将深入剖析类，并解释编译视图的工作原理。

注　意

以前很容易查看早期版本的Razor创建的类，因为每个视图都会在磁盘上生成一个C#文件，然后进行编译以便在应用程序中使用。检查类的内容只是为了找到正确的文件。当前版本的Razor依赖于C#编译器的进步，允许在内存中生成和编译代码，从而提高性能，但很难看到正在执行的事情。要获得以前显示的类，将不得不重新使用ASP.NET Core MVC源代码中包含的一些单元测试，其中提供了Razor依赖于查找和处理视图文件的类的模拟实现。这不是日常开发中需要做的事情，但揭示了视图如何工作。

---

1. 理解类的名称

我们先从Razor创建的类的名称开始。

```
...
```

```
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
...
```

Razor需要使用一些方法来将CSHTML文件的名称和路径转换为分析文件时创建的类，并通过对类名称中的信息进行编码来实现。Razor使用ASPV作为类名前缀，后跟项目名称和控制器名称，最后是视图文件名；这种组合使得当MVC通过本章前面描述的IViewEngine请求视图时，可以轻松检查类是否可用。

2．基类

Razor的很多核心功能，比如能够以@Model的形式引用视图模型，缘于它们派生自基类。

```
...
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
...
```

由于@model指令已用于指定模型类型，因此View类继承自RazorPage类或RazorPage<T>类。RazorPage类提供了让CSHTML文件访问MVC功能的方法和属性，其中有用的RazorPage属性和方法分别如表21-6和表21-7所示。

表**21-6**　视图开发中有用的**RazorPage**属性

属性	描述
Model	返回由操作方法提供的模型数据
ViewData	返回一个ViewDataDictionary对象，该对象提供对其他视图数据功能的访问
ViewContext	返回一个ViewContext对象
Layout	用于指定布局
ViewBag	用于访问ViewBag对象
TempData	提供对Temp数据的访问
Context	返回一个描述当前请求和正在准备的响应的HttpContext对象

User	返回与请求相关联的用户的配置文件

表**21-7**　视图开发中有用的**RazorPage**方法

方法	描述
RenderSection()	用于将视图中的一部分内容插入布局
RenderBody()	用于将不包含在部分（section）中的视图的所有内容插入布局
IsSectionDefined()	用于确定视图是否定义部分的内容

Razor Pages

随着ASP.NET Core 2的发布，微软增加了对Razor Pages的支持，Razor Pages打破了MVC模型，并将支持视图所需的代码与Razor视图相关联。这类似于ASP.NET Web Forms，是微软尝试的一种设计方法，用于重新获得Web Pages平台的简单性，

而不会出现第1章所述的缺点。

不要将本节中描述的RazorPage基类与Razor Pages的功能弄混淆。虽然它们使用相似的名称，但RazorPage基类为MVC框架使用的Razor视图引擎提供了基础。本书没有介绍Razor Pages，因为它不符合MVC模型，也不是MVC平台的一部分。

Razor还提供了一些可以在视图中用于生成内容的辅助属性，如表21-8所示。

表**21-8　Razor**辅助属性

属性	描述
HtmlEncoder	返回一个HtmlEncoder对象，可用于在视图中安全地对HTML内容进行编码
Component	返回一个视图组件助手
Json	返回一个JSON助手

Url	返回一个URL辅助器，可用于使用路由配置生成URL
Html	返回一个HTML助手，可用于生成动态内容。此功能已被标签助手取代，但仍然用于分部视图

表21-6和表21-8中描述的属性在日常MVC开发中访问模型数据、配置视图和执行其他重要任务时会用到。这些属性揭开了Razor的神秘面纱，并将Razor牢牢地置于为人熟知的C#世界中。例如，当使用@Model指令访问视图模型对象或使用@TempData检索临时数据值时，就会用到由RazorPage类定义的属性。

3．理解视图渲染

除了向开发人员提供属性和方法外，RazorPage类还负责通过ExecuteAsyc方法生成响应内容。该方法显示了Razor如何将Index.cshtml文件处理成一组C#语句：

```
...
public override async Task ExecuteAsync() {
    Layout = null;
    WriteLiteral(@"<!DOCTYPE html><html><head>
            <meta name=""viewport"" content=""width=dev
ice-width"" />
            <title>Razor</title>
            <link asp-href-include=""lib/bootstrap/dist
/css/*.min.css""
                rel=""stylesheet"" />
            </head><body class=""m-1 p-1"">This is a li
st of fruit names:");
    foreach (string name in Model) {
        WriteLiteral("<span><b>");
        Write(name);
        WriteLiteral("</b></span>");
    }
    WriteLiteral("</body></html>");
}
...
```

数据值（如Model属性的值）将使用Write方法发送给客户端，该方法会转义字符串，以便浏览器不将其解释为HTML元素。这很重要，因为可以防止恶意数据值向应用程序的输出添加内容。WriteLiteral方法不会转义字符串，并且被用于Index.cshtml文件中的静态内容。当然，浏览器应该

将其解释为HTML元素。结果是CSHTML文件的静态和动态内容包含在常规C#类中，并通过简单的方法发出。

21.4　将动态内容添加到Razor视图中

视图的整个目的是让你能够将域模型的一部分呈现给用户。为此，你需要能够向视图添加动态内容。动态内容在运行时生成，每个请求都可以是不同的。这与你在编写应用程序时创建的对于每个请求都相同的静态内容（例如HTML）相反。可以按照表21-9中描述的不同方式向视图中添加动态内容。

表**21-9**　将动态内容添加到视图中的方式

方式	何时使用
内联代码	用于小型、独立的视图逻辑，如if和foreach语句。这是在视图中创建动态内容的基础工具，其他一些方法便以此为基础。第5章介绍了这种技术，你在以后的章节中也将看到很多这样的例子

标签助手	用于生成HTML元素的属性
部分（section）	用于创建将在特定位置插入布局的内容部分，如本节后面所述
分部视图	用于在视图之间共享视图标记。分部视图可以包含内联代码、HTML辅助方法和其他分部视图的引用。分部视图不会调用操作方法，因此不能用于执行业务逻辑
视图组件	用于创建可重用的UI控件或需要包含业务逻辑的窗口小部件

21.4.1　使用布局部分

Razor视图引擎支持部分（section）的概念，以允许在布局中提供内容区域。部分可以更好地控制将视图的哪些内容插入布局以及放置在哪里。为了演示部分的功能，编辑/Views/Home/Index.cshtml文件，如代码清单21-12所示。

代码清单21-12　在Views/Home文件夹下的Index.cshtml文件中定义部分

```
@model string[]
@{ Layout = "_Layout"; }

@section Header {
    <div class="bg-success">
        @foreach (string str in new [] {"Home", "List",
 "Edit"}) {
            <a class="btn btn-sm btn-primary" asp-actio
n="str">@str</a>
        }
    </div>
}
This is a list of fruit names:
@foreach (string name in Model) {
    <span><b>@name</b></span>
}

@section Footer {
    <div class="bg-success">
        This is the footer
    </div>
}
```

这里从视图中删除了一些HTML元素，并设置了Layout属性，以指定使用名为_Layout.cshtml的布局文件来呈现内容。

这里还在视图中添加了一些部分。部分使用Razor @section表达式定义，后跟部分的名称。此

外创建了标题和页脚部分，内容是HTML标记和Razor表达式的组合，其他示例中已经展示过这些组合。

部分在视图中定义，但应用于带有@RenderSection表达式的布局。为了演示原理，创建Views/Shared文件夹，并添加一个名为_Layout.cshtml的布局文件，内容如代码清单21-13所示。

代码清单21-13　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderSection("Header")
```

```
    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderBody()

    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="bg-info">
        This is part of the layout
    </div>
</body>
</html>
```

当Razor解析布局时，RenderSection辅助程序将被视图中具有指定名称的部分的内容替换。不包含部分的视图内容已使用RenderBody助手插入布局。

可以通过启动应用程序来看到这些部分的效果，如图21-5所示。这里使用一些Bootstrap样式来帮助看清楚哪些部分（section）来自视图，而哪些

来自输出。结果不是很漂亮，但它演示了如何将视图中的内容区域放在布局中的特定位置。

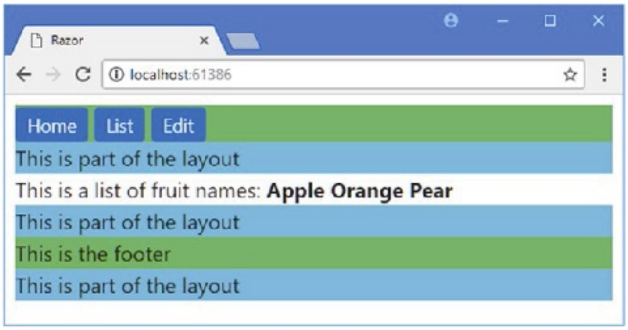


图21-5　使用视图中的部分来定位布局中的内容



注　意

视图只能定义布局中引用的部分。如果尝试在视图中定义布局中没有对应的@RenderSection表达式的部分，MVC将显示异常。

将部分与其余视图混合是不正常的。约定是在视图的开始或结尾定义部分，以便更容易看到哪些内容区域将被视为部分，哪些内容将被RenderBody助手捕获。另一种方式是仅使用部分来定义视图，包括用于正文的部分，如代码清单21-14所示。

代码清单21-14　在Views/Home文件夹下的Index.cshtml文件中根据Razor部分定义视图

```
@model string[]
@{ Layout = "_Layout"; }

@section Header {
    <div class="bg-success">
        @foreach (string str in new [] {"Home", "List",
 "Edit"}) {
            <a class="btn btn-sm btn-primary" asp-actio
n="str">@str</a>
        }
    </div>
}
@section Body {
    This is a list of fruit names:
    @foreach (string name in Model) {
        <span><b>@name</b></span>
```

```
    }
}

@section Footer {
    <div class="bg-success">
        This is the footer
    </div>
}
```

这样做可以提供更清晰的视图，并减少RenderBody捕获无关内容的机会。要使用这种方法，就必须将RenderBody调用替换为RenderBody("body")，如代码清单21-15所示。

代码清单21-15　在Views/Shared文件夹下的_Layout.cshtml文件中将Body渲染为部分

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderSection("Header")
```

```
    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="bg-info">
        This is part of the layout
    </div>
</body>
</html>
```

1．测试部分

可以检查视图是否从布局中定义了特定的部分。如果视图不需要或想要提供特定的内容，可以为部分提供默认内容。修改_Layout.cshtml文件，以检查是否定义了页脚部分，如代码清单21-16所示。

代码清单21-16　在Views/Shared文件夹下的_Layout.cshtml文

件中检查是否定义了某个部分

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderSection("Header")

    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="bg-info">
        This is part of the layout
    </div>

    @if (IsSectionDefined("Footer")) {
        @RenderSection("Footer")
    } else {
        <h4>This is the default footer</h4>
    }

    <div class="bg-info">
        This is part of the layout
    </div>
</body>
```

```
</html>
```

IsSectionDefined助手用于获取要检查的部分的名称，如果在渲染的视图中定义该部分，则返回true。在该例中，可使用该助手来确定在视图未定义页脚部分时是否应该呈现一些默认内容。

2．渲染可选部分

默认情况下，视图必须包含布局中有RenderSection调用的所有部分。如果缺少部分，那么MVC会向用户报告异常。为了演示，为_Layout.cshtml文件添加一个新的RenderSection调用，用于名为scripts的部分，如代码清单21-17所示。

代码清单21-17　在Views/Shared文件夹下的_Layout.cshtml文件中渲染某个不存在的部分

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderSection("Header")

    <div class="bg-info">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="bg-info">
        This is part of the layout
    </div>

    @if (IsSectionDefined("Footer")) {
        @RenderSection("Footer")
    } else {
        <h4>This is the default footer</h4>
    }

    @RenderSection("scripts")

    <div class="bg-info">
        This is part of the layout
    </div>
</body>
</html>
```

当启动应用程序并且Razor引擎尝试渲染布局和视图时，会显示图21-6所示的错误。

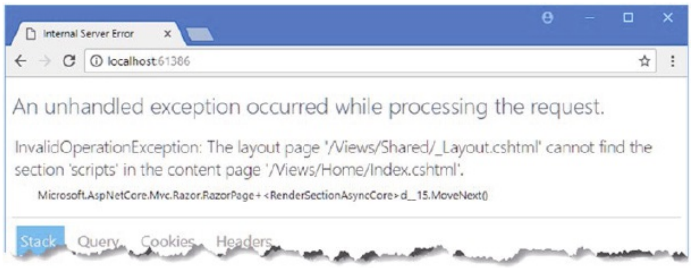


图21-6　当缺少部分时显示错误

可以使用 IsSectionDefined 方法避免对视图未定义的部分进行RenderSection调用，但更好的方法是使用可选部分，也就是向RenderSection方法传递一个额外的虚假参数，如代码清单21-18所示。

代码清单21-18　制作可选部分

```
...
@RenderSection("scripts", false)
...
```

这将创建一个可选部分，如果视图定义了该部

分，就把内容插入结果中；否则，不会引发异常。

21.4.2　使用分部视图

通常需要在应用程序的多个不同位置使用相同的Razor标签和HTML标记。可以使用分部视图，而并非总是将内容复制一份，分部视图是单独的视图文件，其中包含可包含在其他视图中的标记。本节将展示如何创建和使用分部视图，解释它的工作原理，并演示将视图数据传递到分部视图的方法。

1．创建分部视图

分部视图只是常规的CSHTML文件，你需要将它们与常规Razor视图区分开来。Visual Studio为创建预制的分部视图提供了一些支持工具，但创建分部视图的最简单方法是使用MVC View Page模板创建常规视图。为了演示，在Views/Home文件夹中添

加一个名为MyPartial.cshtml的文件，并添加代码清单21-19所示的内容。

代码清单21-19　Views/Home文件夹下的MyPartial.cshtml文件的内容

```
<div class="bg-info">
    <div>This is the message from the partial view.</di
v>
    <a asp-action="Index">This is a link to the Index a
ction</a>
</div>
```

为了演示如何在分部视图中混合使用静态和动态内容，这里定义了一条简单的消息，并添加了一个使用标签助手生成的锚点元素。

2．应用分部视图

可通过从另一个视图中调用@ Html.Partial表达式来使用分部视图。为了演示，在Views/Home文件夹中添加一个名为List.cshtml的新文件，并添加代

码清单21-20所示的内容。

代码清单21-20　Views/Home文件夹下的List.cshtml文件的内容

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    This is the List View
    @Html.Partial("MyPartial")
</body>
</html>
```

Partial方法是一种扩展方法，被应用于添加到Razor从视图文件生成的类的Html属性。这是一个HTML助手，在MVC早期版本中曾是在视图中生成动态内容的方式，但大部分已被标签助手替代。传递给Partial方法的参数是分部视图的名称，分部视

图的内容被插入发送到客户端的输出。

提　示

Razor查找分部视图与查看常规视图的方式相同（搜索Views/<controller>和Views/Shared文件夹），这意味着可以创建特定于特定控制器的专用分部视图，并覆盖Shared文件夹中同名的分部视图。

可以通过启动应用程序并导航到URL /Home/List来查看分部视图的使用效果，如图21-7所示。

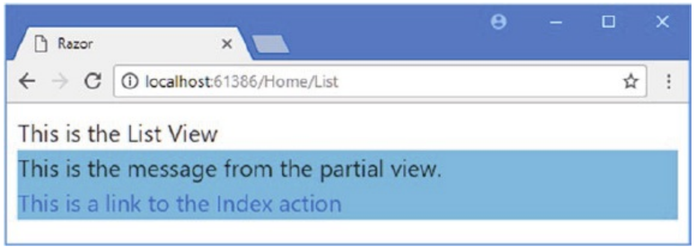


图21-7　使用分部视图

3．使用强类型的分部视图

可以创建强类型的分部视图，并提供在呈现分部视图时使用的视图模型对象。为了演示此功能，可在Views/Home文件夹中创建一个新的名为MyStronglyTypedPartial.cshtml的视图文件并添加代码清单21-21所示的内容。

代码清单21-21　Views/Home文件夹下的MyStronglyTypedPartial.cshtml文件的内容

```
@model IEnumerable<string>

<div class="bg-info">
    This is the message from the partial view.
    <ul>
        @foreach (string str in Model) {
            <li>@str</li>
```

```
        }
    </ul>
</div>
```

视图模型类型使用标准@model表达式来定义，这里使用@foreach循环将视图模型对象的内容显示为HTML代码清单中的条目。为了演示如何使用这个分部视图，更新/Views/Common/List.cshtml文件，如代码清单21-22所示。

代码清单21-22　Views/Common文件夹下的List.cshtml文件的内容

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    This is the List View
    @Html.Partial("MyStronglyTypedPartial",
        new string[] { "Apple", "Orange", "Pear" })
```

```
</body>
</html>
```

与上一个示例的区别在于，这里将一个额外的参数传递给提供视图模型的Partial辅助方法。可以通过启动应用程序和导航到URL——/Home/List来查看使用的强类型的分部视图，如图21-8所示。

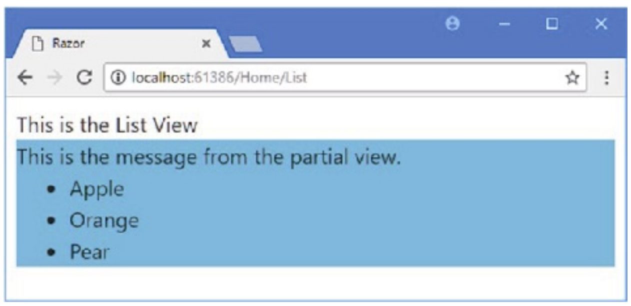


图21-8　使用强类型的分部视图

21.4.3　将JSON内容添加到视图中

视图中经常包含JSON，以便为客户端JavaScript代码提供可用于动态生成内容的数据。为了准备这个例子，可通过编辑bower.json文件将jQuery包添加到应用程序中，如代码清单21-23所

示，这将使浏览器更容易处理作为HTML文档的一部分接收的JSON数据。

代码清单21-23　在bower.json文件中添加jQuery

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

代码清单21-24显示了List.cshtml视图文件的新增功能，该视图使用Razor在发送到浏览器的响应中包含JSON数据。

代码清单21-24　在Views/Common文件夹下的List.cshtml文件中使用JSON数据

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
```

```
    <meta name="viewport" content="width=device-width"
/>
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
    <script id="jsonData" type="application/json">
        @Json.Serialize(new string[] { "Apple", "Orange
", "Pear" })
    </script>
</head>
<body class="m-1 p-1">
    This is the List View
    <ul id="list"></ul>
</body>
</html>
```

@Json.Serialize表达式接收一个对象并将其序列化为JSON格式。以上代码已经在包含JSON数据的视图中添加了一个script元素。当视图被呈现并发送到浏览器时，其中包含如下元素：

```
...
<script id="jsonData" type="application/json">["Apple",
"Orange","Pear"]</script>
...
```

为了利用JSON数据，代码清单21-25添加了jQuery库和一些JavaScript代码，这些代码使用

jQuery来解析JSON数据并动态创建一些HTML元素。

代码清单21-25　在Views/Common文件夹下的List.cshtml文件中使用JSON数据

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
    <script id="jsonData" type="application/json">
        @Json.Serialize(new string[] { "Apple", "Orange
", "Pear" })
    </script>
    <script asp-src-include="lib/jquery/dist/*.min.js">
</script>
    <script type="text/javascript">
        $(document).ready(function () {
            var list = $("#list")
            JSON.parse($("#jsonData").text()).forEach(f
unction (val) {
                console.log("Val: " + val);
                list.append($("<li>").text(val));
            });
        });
    </script>
```

```
</head>
<body class="m-1 p-1">
    This is the List View
    <ul id="list"></ul>
</body>
</html>
```

如果运行示例应用程序并请求URL /Home/List，就会显示图21-9所示的内容。这不是JSON数据最重要的使用方法，但它显示了如何将JSON数据包含在视图中。

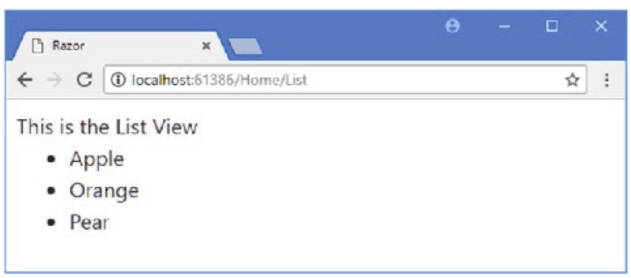


图21-9　在视图中使用JSON数据

21.5　配置Razor

可以使用Microsoft.AspNetCore.Mvc.Razor命名空间中定义的RazorViewEngineOptions类来配置

Razor。该类定义了两个配置属性，如表21-10所示。

表**21-10　RazorViewEngineOptions**类定义的配置属性

配置属性	描述
FileProvider	用于设置为文件和目录提供Razor的对象，由Microsoft.AspNetCore.FileProviders.IFileProvider接口定义，默认由从磁盘读取文件的PhysicalFileProvider实现
ViewLocationExpanders	用于配置视图扩展，用于更改Razor查找视图的方式

提　示

如果想深入学习，那么可以通过创建实现了Microsoft.AspNetCore.Mvc.Razor命名空间中接口的类来替换内部Razor组件，并注册到Startup类中的

service provider。但大多数开发人员并不需要这么做，但如果希望完全控制应用程序中内容的处理方式，那么这是一个很有用的方式。可以从GitHub下载Razor的源代码作为开始。

许多应用程序不需要更改FileProvider属性，因为从磁盘读取视图文件正是大多数项目所需要的，而Razor仅使用service provider来加载视图，以便在应用程序首次运行时进行编译。ViewLocationExpanders属性更有用，因为它允许应用程序将自定义逻辑应用于Razor查看视图的方式。

视图位置扩展

Razor使用视图位置扩展来构建应该搜索视图位置的清单。视图位置扩展实现了

IViewLocationExpander接口，该接口的定义如下：

```
using System.Collections.Generic;

namespace Microsoft.AspNetCore.Mvc.Razor {

    public interface IViewLocationExpander {

        void PopulateValues(ViewLocationExpanderContext
 context);

        IEnumerable<string> ExpandViewLocations(ViewLoc
ationExpanderContext context,
            IEnumerable<string> viewLocations);
    }
}
```

接下来解释视图位置扩展的工作原理，并创建IViewLocationExpander接口的自定义实现。为了准备创建视图位置扩展，代码清单21-26已经更改了Home控制器的Index操作方法，以便请求不存在的视图。视图不存在的错误消息将显示Razor搜索视图的位置以及视图位置扩展带来的影响。

代码清单21-26　在Controllers文件夹下的HomeController.cs文

件中请求不存在的视图

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Views.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() =>
            View("MyView", new string[] { "Apple", "Ora
nge", "Pear" });

        public ViewResult List() => View();
    }
}
```

如果启动应用程序并请求默认URL，你将看到错误消息中显示的默认视图搜索位置，如下所示：

```
/Views/Home/MyView.cshtml
/Views/Shared/MyView.cshtml
```

1．创建简单的视图位置扩展

最简单的视图位置扩展只需要更改Razor查找视图的位置集合。这可以通过实现

ExpandViewLocations方法并返回要支持的位置清单来完成。为了演示，在Infrastructure文件夹中添加一个名为SimpleExpander.cs的类文件，内容如代码清单21-27所示。

代码清单21-27　Infrastructure文件夹下的SimpleExpander.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views.Infrastructure {

    public class SimpleExpander : IViewLocationExpander
 {

        public void PopulateValues(ViewLocationExpander
Context context) {
            // do nothing - not required
        }

        public IEnumerable<string> ExpandViewLocations(
                ViewLocationExpanderContext context,
                IEnumerable<string> viewLocations) {

            foreach (string location in viewLocations)
{
                yield return location.Replace("Shared",
 "Common");
```

```
            }
            yield return "/Views/Legacy/{1}/{0}/View.cs
html";
        }
    }
}
```

当需要搜索位置清单时，Razor会调用ExpandViewLocations方法，并且使用viewLocations参数中的一系列字符串提供默认位置。位置表示为带有占位符的模板，占位符用于指代操作方法和控制器的名称。以下是在不使用路由区域的应用程序中默认使用的位置模板：

```
"/Views/{1}/{0}.cshtml"
"/Views/Shared/{0}.cshtml"
```

占位符{0}用于指代操作方法的名称，占位符用于指代{1}控制器的名称。视图位置扩展的作用是返回应被搜索的位置集合，可使用string.Replace方法在默认位置中更改Shared为Common，并添加自己的位置，位置将遵循不同的文件和文件夹结

构。

代码清单21-28通过在Startup类中配置Razor来设置视图位置扩展。ViewLocationExpanders属性会返回一个在其中调用Add方法的List<IViewLocationExpander>对象。

代码清单21-28　在Startup.cs文件中配置Razor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
            services.Configure<RazorViewEngineOptions>(
options => {
                options.ViewLocationExpanders.Add(new S
```

```
impleExpander());
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

如果运行示例，错误消息将显示自定义的视图位置扩展提供给Razor的位置集合。

```
/Views/Home/MyView.cshtml
/Views/Common/MyView.cshtml
/Views/Legacy/Home/MyView/View.cshtml
```

2．为请求选择特定视图

视图位置扩展可以轻松更改所有请求的检索位置，也可以更改单个请求的搜索位置。前面的例子仅实现了ExpandViewLocations方法，但实际作用来

自PopulateValues方法，这是IViewLocationExpander接口中的另一个方法。

Razor每次需要一个视图时，就会调用视图位置扩展的PopulateValues方法，为上下文数据提供一个ViewLocationExpanderContext对象。表21-11显示了ViewLocationExpanderContext类定义的属性。

表**21-11　ViewLocationExpanderContext**类定义的属性

属性	描述
ActionContext	返回一个ActionContext对象，用于描述已请求视图的操作方法，并包含有关请求和响应的详细信息
ViewName	返回操作方法请求的视图的名称
ControllerName	返回包含操作方法的控制器的名称
AreaName	如果已定义区域，就返回包含控制器的区域的名称
IsMainPage	如果Razor正在寻找分部视图，就返回false；否则，返回true
	返回一个IDictionary<string,string>对象，视图位置扩展会添加唯一标识请

Values	求类别的键/值对，具体将在下面的内容中介绍

PopulateValues方法的目的是通过为Context对象的Values属性返回的字典添加键/值对来对请求分类。Razor不管请求如何分类，用于填充字典的方法都完全留给视图位置扩展，这很容易通过一个例子来解释。将一个名为ColorExpander.cs的类文件添加到Infrastructure文件夹中，并用它定义代码清单21-29所示的类。

代码清单21-29　Infrastructure文件夹下的ColorExpander.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views.Infrastructure {

    public class ColorExpander : IViewLocationExpander
{
        private static Dictionary<string, string> Color
s
            = new Dictionary<string, string> {
                ["red"] = "Red", ["green"] = "Green", [
"blue"] = "Blue"
```

```
            };

        public void PopulateValues(ViewLocationExpander
Context context) {

            var routeValues = context.ActionContext.Rou
teData.Values;
            string color;

            if (routeValues.ContainsKey("id")
                    && Colors.TryGetValue(routeValues["
id"] as string, out color)
                    && !string.IsNullOrEmpty(color)) {
                context.Values["color"] = color;
            }
        }

        public IEnumerable<string> ExpandViewLocations(
                ViewLocationExpanderContext context,
                IEnumerable<string> viewLocations) {
            string color;
            context.Values.TryGetValue("color", out col
or);
            foreach (string location in viewLocations)
{
                if (!string.IsNullOrEmpty(color)) {
                    yield return location.Replace("{0}"
, color);
                } else {
                    yield return location;
                }
            }
        }
    }
}
```

PopulateValues方法使用ActionContext获取路由数据，并查找URL参数片段id的值。如果有一个id片段的值为红色、绿色或蓝色，视图位置扩展就会向值字典添加Color属性。在分类的过程中，id片段匹配颜色的请求可使用颜色键进行分类，颜色键的值由参数片段的值定义。

接下来，Razor调用ExpandViewLocations方法并提供与PopulateValues方法相同的Context对象。该操作允许视图位置扩展查看先前执行的分类，并生成Razor应该查看的视图的位置集合。以上示例使用string.Replace方法来替换具有颜色名称的{0}占位符。

提　示

Razor为每个视图请求调用PopulateValues方法，但缓存由ExpandViewLocations方法返回的搜索位置集合。这意味着PopulateValues方法生成的同一组分类键和值的后续请求不需要调用ExpandViewLocations方法。

代码清单21-30已将Razor配置为使用ColorExpander类。

代码清单21-30　在Views文件夹下的Startup.cs文件中添加视图位置扩展

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
            services.Configure<RazorViewEngineOptions>(
options => {
                options.ViewLocationExpanders.Add(new S
impleExpander());
                options.ViewLocationExpanders.Add(new C
olorExpander());
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

通过启动应用程序并请求URL /Home/Index/red可以查看新的视图位置扩展的效果，并使Razor同时在以下位置进行搜索：

```
/Views/Home/Red.cshtml
/Views/Common/Red.cshtml
/Views/Legacy/Home/Red/View.cshtml
```

类似地，对URL /Home/Index/green的请求会使Razor在以下位置进行搜索：

```
/Views/Home/Green.cshtml
/Views/Common/Green.cshtml
/Views/Legacy/Home/Green/View.cshtml
```

视图位置扩展的注册顺序很重要，因为一个视图位置扩展的ExpandViewLocations方法生成的位置集合，会被用作清单中下一个视图位置扩展的viewLocations参数。从之前显示的位置可以看到这一点，其中，Views/Common和Views/Legacy位置由SimpleExpander类生成，该类出现在Startup类中的ColorExpander之前。

21.6　小结

本章演示了如何创建自定义视图引擎，并通过将CSHTML文件转换为C#类来解释Razor的工作原理。本章还展示了如何使用布局部分（section）和分部视图，并讨论了如何更改Razor用于查找视图文件的位置。下一章将介绍用于提供支持分部视图的逻辑的视图组件。

第22章　视图组件

本章描述视图组件，这是ASP.NET Core MVC新增的功能，以替代以前版本的子action功能。视图组件是提供操作样式逻辑以支持分部视图的类，这意味着可以将复杂内容嵌入视图，同时可以方便维护C#代码和支持单元测试。表22-1介绍了视图组件的背景。

表**22-1**　视图组件的背景

问题	答案
什么是视图组件？	视图组件是提供应用程序逻辑以支持分部视图或将小部分的HTML或JSON数据注入父视图的类
视图组件有什么作用？	没有视图组件，将难以创建易于维护和支持单元测试的功能，如购物车或登录面板
如何使用视图组件？	视图组件通常派生自ViewComponent类，并使用@await Component.InvokeAsync表达式应用于父视图

在使用中有什么陷阱或局限性？	没有，视图组件能提供简单可靠的功能，如果不使用视图组件，视图中的应用程序逻辑将会难以维护和测试
是否有其他替代方法？	可以将数据访问和处理逻辑直接放在分部视图中，但是结果很难使用，难以有效测试

表22-2列出了本章要介绍的操作。

表**22-2**　本章要介绍的操作

操作	方法	代码清单
使用自己的逻辑和数据提供分部视图	使用视图组件	代码清单22-12
调用视图组件	在视图中使用@await Component.InvokeAsync表达式	代码清单22-13
简化对上下文数据和结果的访问	从ViewComponent类派生	代码清单22-14～代码清单22-16
选择分部视图	使用View方法创建并返回ViewViewComponentResult对象	代码清单22-17～代码清单22-

		19
创建HTML片段	如果不想要对HTML片段进行编码，那么返回Content方法创建的ContentViewComponentResult对象或者明确地创建HtmlContentViewComponentResult对象	代码清单22-20和代码清单22-21
使用请求的详细信息生成结果	使用视图组件上下文数据	代码清单22-22
在调用视图组件时提供上下文数据	提供InvokeAsync方法的参数	代码清单22-23～代码清单22-25
创建异步视图组件	实现InvokeAsync方法并返回Tast对象以得到想要的结果	代码清单22-26～代码清单22-29
创建混合的控制器/视图组件	将ViewComponent属性应用于控制器类	代码清单22-30～代码清单22-33

22.1　准备示例项目

在本章中，将使用ASP.NET Core Web Application（.NET Core）模板创建一个名为UsingViewComponents的Empty项目。

22.1.1　创建模型和存储库

这里需要两个不同的数据源来演示视图组件的工作原理。为了使部分应用程序对一组Product对象进行操作，创建Models文件夹，并添加一个名为Product.cs的文件，用来定义代码清单22-1所示的类。

代码清单22-1　Models文件夹下的Product.cs文件的内容

```
namespace UsingViewComponents.Models {

    public class Product {
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

要为Product对象创建存储库，可将一个名为

ProductRepository.cs的文件添加到Models文件夹中，并定义代码清单22-2所示的接口和实现类。

代码清单22-2　Models文件夹下的ProductRepository.cs文件的内容

```
using System.Collections.Generic;

namespace UsingViewComponents.Models {

    public interface IProductRepository {
        IEnumerable<Product> Products { get; }
        void AddProduct(Product newProduct);
    }

    public class MemoryProductRepository : IProductRepo
sitory {
        private List<Product> products = new List<Produ
ct> {
                new Product { Name = "Kayak", Price = 2
75M },
                new Product { Name = "Lifejacket", Pric
e = 48.95M },
                new Product { Name = "Soccer ball", Pri
ce = 19.50M }
        };

        public IEnumerable<Product> Products => product
s;

        public void AddProduct(Product newProduct) {
```

```
            products.Add(newProduct);
        }
    }
}
```

IProductRepository接口定义了一组有限的存储库功能，MemoryProductRepository类使用List实现了该接口。为了使应用程序的其他部分将对City进行操作，在Models文件夹中添加一个名为City.cs的类文件，并用它定义代码清单22-3所示的类。

代码清单22-3　Models文件夹下的City.cs文件的内容

```
namespace UsingViewComponents.Models {

    public class City {
        public string Name { get; set; }
        public string Country { get; set; }
        public int Population { get; set; }
    }
}
```

对于City对象的存储库，创建一个名为CityRepository.cs的类文件，并用它定义代码清单22-4所示的接口和实现类。

代码清单22-4　Models文件夹下的CityRepository.cs文件的内容

```
using System.Collections.Generic;

namespace UsingViewComponents.Models {

    public interface ICityRepository {
        IEnumerable<City> Cities { get; }

        void AddCity(City newCity);
    }

    public class MemoryCityRepository : ICityRepository
 {

        private List<City> cities = new List<City> {
            new City { Name = "London", Country = "UK",
 Population = 8539000},
            new City { Name = "New York", Country = "US
A", Population = 8406000 },
            new City { Name = "San Jose", Country = "US
A", Population = 998537 },
            new City { Name = "Paris", Country = "Franc
e", Population = 2244000 }
        };

        public IEnumerable<City> Cities => cities;

        public void AddCity(City newCity) {
            cities.Add(newCity);
        }
    }
}
```

ICityRepository接口提供了一组有限的存储库功能，MemoryCityRepository类使用List来实现接口。

22.1.2　创建控制器和视图

由于这里只需要一个控制器，因此创建Controllers文件夹，将一个名为HomeController.cs的文件添加到Controllers文件夹中，并用它定义代码清单22-5所示的类。

代码清单22-5　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Controllers {

    public class HomeController : Controller {
        private IProductRepository repository;

        public HomeController(IProductRepository repo)
{
            repository = repo;
```

```
        }
        public ViewResult Index() => View(repository.Pr
oducts);

        public ViewResult Create() => View();

        [HttpPost]
        public IActionResult Create(Product newProduct)
 {
            repository.AddProduct(newProduct);
            return RedirectToAction("Index");
        }
    }
}
```

Home控制器使用构造函数来声明对IProductRepository接口的依赖关系，当控制器用于处理请求时，service provider将解析该接口。Index操作方法从存储库中检索所有Product对象，并使用默认视图进行渲染。两个Create方法使用的是Post/Redirect/Get方式，用客户端提供的表单数据生成新的对象并添加到存储库中。

该例中的视图将共享一种通用布局。创建Views/Shared文件夹，并添加一个名为

_Layout.cshtml的文件，内容如代码清单22-6所示。

代码清单22-6　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                <div class="bg-info text-center m-1 p-1
">City Placeholder</div>
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

以上布局定义了标题，其中包含一个占位符，用于显示本章后面使用City存储库创建的内容。创

建Views/Home文件夹，并添加一个名为Index.cshtml的视图文件，内容如代码清单22-7所示，其中列出了表格中Product对象的详细信息。

代码清单22-7　Views/Home文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<Product>
@{
    ViewData["Title"] = "Products";
    Layout = "_Layout";
}
<table class="table table-sm table-striped table-border
ed">
    <thead>
        <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody>
        @foreach (var product in Model) {
            <tr>
                <td>@product.Name</td>
                <td>@product.Price</td>
            </tr>
        }
    </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</
a>
```

Index视图中的最后一个元素被定义为按钮，且链接指向Create操作方法，用来在存储库中新建一个Product对象。要提供用户填写的表单，可在Views/Home文件夹新建Create.cshtml文件，并添加代码清单22-8所示的内容。

代码清单22-8　Views/Home文件夹下的Create.cshtml文件的内容

```
@model Product
@{
    ViewData["Title"] = "Create Product";
    Layout = "_Layout";
}

<form method="post" asp-action="Create">
    <div class="form-group">
        <label asp-for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price">Price:</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Creat
e</button>
    <a class="btn btn-secondary" asp-action="Index">Can
cel</a>
```

```
</form>
```

这些视图需要使用内置的标签助手，可通过在Views文件夹中创建_ViewImports.cshtml文件并添加如代码清单22-9所示的表达式来启用标签助手，这可以使Models文件夹中的类在没有命名空间的情况下也可用。

代码清单22-9　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using UsingViewComponents.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

这些视图还依赖于Bootstrap CSS包来提供样式。这里使用Bower Configuration File模板在项目的根目录中创建bower.json文件，并将Bootstrap添加到dependencies部分，如代码清单22-10所示。

代码清单22-10　在UsingViewComponents文件夹下的bower.json文件中添加Bootstrap

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

22.1.3　配置应用程序

最后的准备步骤是配置应用程序，如代码清单22-11所示。除了配置MVC服务和中间件之外，还为两个数据存储库创建了单例服务。

代码清单22-11　UsingViewComponents文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using UsingViewComponents.Models;

namespace UsingViewComponents {
```

```
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IProductRepository, MemoryProductRepository>();
            services.AddSingleton<ICityRepository, MemoryCityRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

如果运行应用程序，你将看到Product存储库中的Product对象列表。可以通过单击Create按钮、填写表单并提交到服务器来添加新的商品，添加完成后链接会重定向到商品列表，如图22-1所示。由于应用程序中的视图使用共享布局，因此整个过程中会显示城市数据的占位符。

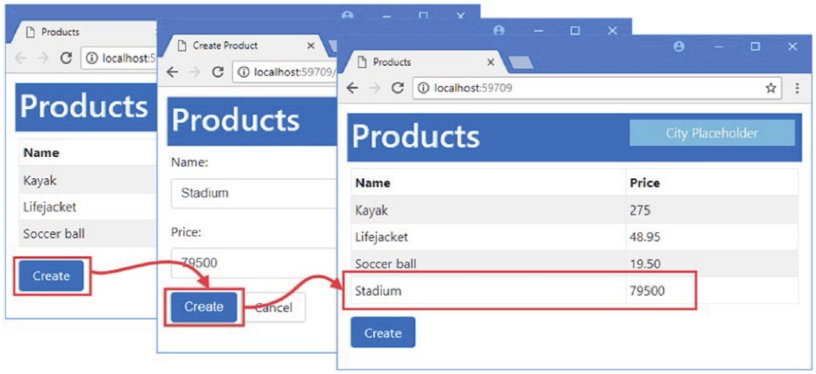


图22-1　运行示例应用程序

22.2　视图组件

应用程序通常需要在视图中嵌入一些与视图主要功能无关的内容。常见的示例包括网站导航工具、以及让用户不用访问单独的页面就能登录的身份验证面板等。

所有这些示例的共同线索是显示嵌入式内容所需的数据不是从操作传递到视图的模型数据的一部分。因此，这里在示例应用程序中创建了两个存储库。使用City存储库生成的一些内容，将很难从Product存储库的用于接收数据的操作中获取。

第21章描述了如何使用分部视图来创建视图中需要的可重用标记，避免在应用程序的多个位置复制相同的内容。分部视图虽然有用，但它们只包含HTML和Razor指令的片段，并且操作的数据是从父视图接收的。如果需要显示不同的数据，就会遇到问题。可以直接从分部视图访问需要的数据，但这会破坏支持MVC模式的关注点分离原则，并导致将数据检索和处理逻辑放置在不能进行单元测试的视图文件中。也可以扩展应用程序使用的视图模型，以使其包含所需的数据，但这意味着必须更改每个操作方法，并使对操作方法的测试无法被隔离。

这正是需要使用视图组件的地方，视图组件是C#类，提供了包含所需数据的分部视图，独立于父视图和操作。在这方面，视图组件负责专门的操作，但是仅用于提供数据的分部视图，不能接收HTTP请求，并且提供的内容将始终包含在父视图

中。

22.3　创建视图组件

视图组件可以通过3种不同的方式创建，分别是通过定义POCO视图组件、通过从ViewComponent基类派生以及使用ViewComponent特性。本节介绍前两种方式，22.4节将介绍第三种方式。

22.3.1　创建POCO视图组件

POCO视图组件是一种不依赖于任何MVC API来提供视图组件功能的类。与POCO控制器一样，这种视图组件非常难以使用，但有助于了解它们的工作原理。POCO视图组件是任何一种名称以ViewComponent结尾并定义了Invoke方法的类。视图组件类可以在应用程序中的任何位置进行定义，

但是约定将它们组合在名为Components的文件夹中，该文件夹位于项目的根目录下。创建这个文件夹，并添加一个名为PocoViewComponent.cs的类文件，定义代码清单22-12所示的类。

代码清单22-12　Components文件夹下的PocoViewComponent.cs文件的内容

```
using System.Linq;
using UsingViewComponents.Models;

namespace UsingViewComponents.ViewComponents {

    public class PocoViewComponent {
        private ICityRepository repository;

        public PocoViewComponent(ICityRepository repo)
{
            repository = repo;
        }

        public string Invoke() {
            return $"{repository.Cities.Count()} cities
, "
                + $"{repository.Cities.Sum(c => c.Popul
ation)} people";
        }
    }
}
```

视图组件可以利用依赖注入来接收所需的服务。在此例中，POCO视图组件声明了对ICityRepository接口的从属关系，然后在Invoke方法中使用该接口来创建描述城市数量和人口总数的字符串。

为了使用视图组件，需要使用Razor @await Component.Invoke表达式。可通过提供类的名称（不包含末尾的ViewComponent）作为参数来选择视图组件。代码清单22-13已经删除了共享布局中的占位符，而使用POCO视图组件取代。

代码清单22-13　在Views/Shared文件夹下的_Layout.cshtml文件中使用视图组件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
```

```
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                @await Component.InvokeAsync("Poco")
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

为了应用视图组件，指定Poco作为Invoke方法的参数。当视图使用布局时，将定位PocoViewComponent类，调用Invoke方法，并将结果插入父视图的输出，如图22-2所示。

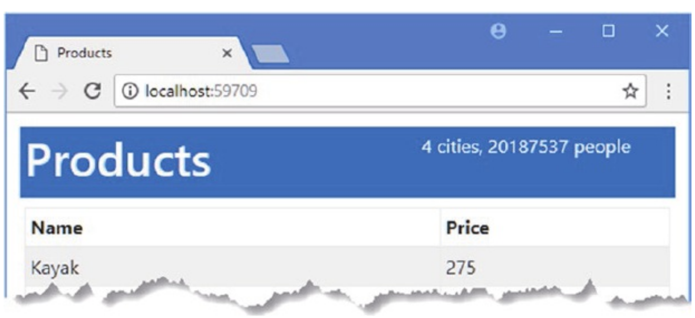


图22-2　使用简单的视图组件

这只是一个简单的例子，但它说明了视图组件的一些重要特性。首先，PocoViewComponent类能够访问所需的数据，而不依赖于处理HTTP请求或父视图的操作。其次，在C#类中定义获取和处理城市信息所需的逻辑意味着可以轻松进行单元测试（参见22.3.4节）。最后，应用程序没有分裂，视图在关注Product对象的视图模型中包含City对象。

警　告

在视图中应用视图组件时，必须包含await关键字。如果只调用@Component.Invoke，虽然不会报错，但只会显示任务的字符串表示形式，类似于System.Threading.Tasks.Task`1

[Microsoft.AspNetCore.Html.IHtmlContent]。

22.3.2　从ViewComponent基类派生

POCO视图组件的功能有限，除非它们可以利用MVC API，这就需要执行一些更多的操作，需要从ViewComponent基类派生。Microsoft.AspNetCore.Mvc命名空间中定义的ViewComponent类可以方便地访问上下文数据，从而更容易生成结果。代码清单22-14显示了添加到Components文件夹下的CitySummary.cs文件的内容。

代码清单22-14　Components文件夹下的CitySummary.cs文件的内容

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
```

```
namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
        private ICityRepository repository;

        public CitySummary(ICityRepository repo) {
            repository = repo;
        }

        public string Invoke() {
            return $"{repository.Cities.Count()} cities
, "
            + $"{repository.Cities.Sum(c => c.Populatio
n)} people";
        }
    }
}
```

在从基类派生时，不需要在类名中包含ViewComponent。除了使用基类之外，POCO视图组件与POCO功能相同。下面将展示如何使用基类提供的便利功能来实现不同的视图组件功能。

提　示

注意，代码清单22-14没有重写Invoke方法。ViewComponent类不提供Invoke方法的默认实现，因而必须明确定义。

为了演示视图组件的特性，更改共享布局中使用的组件，如代码清单22-15所示，使用nameof而不是直接用字符串指定视图组件的名称，第4章介绍过，这样可以减少类名错误的可能性。

代码清单22-15　在Views/Shared文件夹下的_Layout.cshtml文件中修改视图组件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
```

```
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                @await Component.InvokeAsync(nameof(CitySummary))
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

修改视图导入文件，从而可以在没有命名空间的nameof表达式中引用CitySummary类，如代码清单22-16所示。

代码清单22-16　在Views文件夹下的_ViewImports.cshtml文件中添加命名空间

```
@using UsingViewComponents.Models
@using UsingViewComponents.Components
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

22.3.3　视图组件结果

将简单的字符串值插入父视图的功能并不是特别有用，但幸运的是，视图组件能够做更多。通过使Invoke方法返回实现了IViewComponentResult接口的对象，可以实现更复杂的效果。3个内置的类实现了IViewComponentResult接口，如表22-3所示，ViewComponent基类提供了创建它们的便利方法。接下来介绍每种结果类型的用法。

注　意

如果使用POCO视图组件，就可以直接创建这些类的实例，尽管它们可能很难处理，因为它们都具有很复杂的构造函数。ViewComponent类则提供了创建它们的简便方法。

表**22-3**　内置的**IViewComponentResult**实现类

名称	描述
ViewViewComponentResult	用于指定Razor视图，并且具有可选的视图模型数据。该类的实例可使用View方法创建
ContentViewComponentResult	用于指定包含在HTML文档中且被安全编码的文本结果。该类的实例可使用Content方法创建
HtmlContentViewComponentResult	用于指定包含在HTML文档中的HTML片段，无须进一步编码。没有可用于创建这种类型结果的ViewComponent方法

有两种结果类型需要做特殊处理。如果视图组件返回一个字符串，那么它用于创建ContentViewComponentResult对象。如果视图组件返回IHtmlContent对象，那么它用于创建HtmlContentViewComponentResult对象。

1．返回分部视图

最常使用的响应为ViewViewComponentResult对象，它告诉Razor呈现局部视图，并将结果包含在父视图中。ViewComponent基类提供了用于创建ViewViewComponentResult对象的View方法，并且有4个版本的View方法可用，如表22-4所示。

表**22-4　ViewComponent.View**方法

名称	描述
View()	为视图组件选择默认视图，并且不提供视图模型
View(model)	选择默认视图并使用指定的对象作为视图模型
View(viewName)	可以选择指定的视图，而不提供视图模型
View(viewName,model)	选择指定的视图并使用指定的对象作为视图模型

这些View方法对应于由Controller基类提供的方法，并且以相同的方式使用。在Models文件夹中添加一个名为CityViewModel.cs的类文件，用来定

义视图模型，如代码清单22-17所示。

代码清单22-17　Models文件夹下的CityViewModel.cs文件的内容

```
namespace UsingViewComponents.Models {

    public class CityViewModel {
        public int Cities { get; set; }
        public int Population { get; set; }
    }
}
```

代码清单22-18修改了CitySummary视图组件的Invoke方法，以便能够使用View方法来选择分部视图，并使用CityViewModel对象提供视图数据。

代码清单22-18　在Controller文件夹下的CitySummary.cs文件中选择分部视图

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
        private ICityRepository repository;
```

```
        public CitySummary(ICityRepository repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            return View(new CityViewModel{
                Cities = repository.Cities.Count(),
                Population = repository.Cities.Sum(c =>
 c.Population)
            });
        }
    }
}
```

在视图组件中选择分部视图与在控制器中选择视图类似，但有两个重要区别：Razor在不同位置查找视图，并且如果未指定视图，就使用不同的默认视图名称查找视图。

因为没有为视图组件创建分部视图，所以在运行显示Razor所要查找的文件的应用程序时，你将会看到一条错误消息。

如果没有指定名称，那么Razor会尝试寻找名

为Default.cshtml的文件。Razor会在两个位置（/Views/Home/Components/CitySummary/Default.cshtml与/Views/Shared/Components/CitySummary/Default.cs查找分部视图。第一个位置考虑了处理HTTP请求的控制器的名称，以允许每个控制器拥有自己的自定义视图；第二个位置可在所有控制器之间共享。

提　示

请注意，共享的分部视图仍然由视图组件区分，这意味着视图组件不能共享分部视图。可以通过在调用View方法时，在视图名称中包含路径来覆盖这种行为，比如调用

View（"Views/Shared/Components/Common/Default.html"）将覆盖分部视图的默认搜索位置。

为完成此例，创建Views/Home/Components/CitySummary文件夹，并添加一个名为Default.cshtml的视图文件，内容如代码清单22-19所示。

代码清单22-19　Views/Home/Components/CitySummary文件夹下的Default.cshtml文件的内容

```
@model CityViewModel

<table class="table table-sm table-bordered">
    <tr>
        <td>Cities:</td>
        <td class="text-right">
            @Model.Cities
        </td>
    </tr>
    <tr>
        <td>Population:</td>
        <td class="text-right">
            @Model.Population.ToString("#,###")
```

```
        </td>
    </tr>
</table>
```

视图组件的分部视图与控制器的工作方式相同。在这种情况下，创建一个强类型的视图，它可以包含一个CityViewModel对象，并在表格中显示Cities和Population值，如图22-3所示。

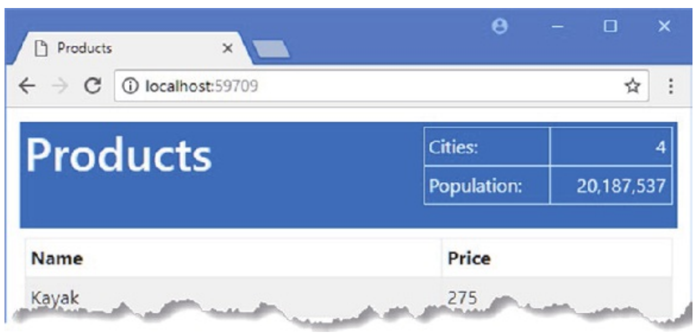


图22-3　使用视图组件渲染视图

2．返回HTML片段

ContentViewComponentResult类可以在不使用视图的情况下，在父视图中包含HTML片段。ContentViewComponentResult类的实例可使用继承

}

自ViewComponent基类的Content方法创建，该方法接收字符串值。代码清单22-20演示了如何使用Content方法。除了Content方法之外，Invoke方法可以返回一个字符串，MVC将自动把它转换为ContentViewComponentResult对象。

代码清单22-20　在Components文件夹下的CitySummary.cs文件中使用Content方法

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
        private ICityRepository repository;
        public CitySummary(ICityRepository repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            return Content("This is a <h3><i>string</i>
</h3>");
        }
    }
}
```

对Content方法接收到的字符串进行编码，使其安全地包含在HTML文档中。当处理由用户或外部系统提供的内容时，这特别重要，因为可以阻止JavaScript内容被嵌入应用程序生成的HTML中。在此例中，传递给Content方法的字符串包含一些基本的HTML标签，如果运行应用程序，你将看到它们已被安全地编码，如图22-4所示。



图22-4　使用视图组件返回编码的HTML片段

如果查看视图组件生成的HTML，你会发现尖括号已被替换，以使浏览器不将内容解释为HTML元素，如下所示：

```
...
<div class="col-5">This is a &lt;h3><i>string&lt;/i></h
```

```
3></div>
...
```

如果信任它们的来源并希望解释为HTML，就不需要对内容进行编码。Content方法总是对参数进行编码，因此必须直接创建HtmlContentViewComponentResult对象，并为其构造函数提供一个HtmlString对象，该对象会显示你认为可以安全显示的字符串，因为来源值得信任，或者因为你确信它们已经编码，如代码清单22-21所示。

代码清单22-21　在Components文件夹下的CitySummary.cs文件中返回信任的HTML片段

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Html;

namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
```

```
        private ICityRepository repository;

        public CitySummary(ICityRepository repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            return new HtmlContentViewComponentResult(
                new HtmlString("This is a <h3><i>string
</i></h3>"));
        }
    }
}
```

以上方法应慎用，只有当使用不能篡改的内容来源，并执行自己的编码时才可使用。如果运行应用程序，你将看到尖括号已包含在父视图中而未修改，这允许浏览器将视图组件的输出解释为HTML元素，如图22-5所示。

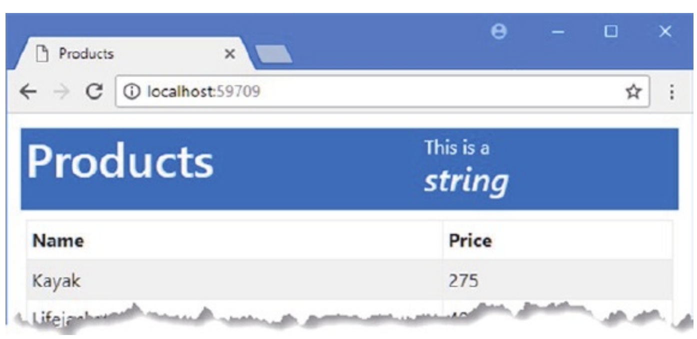


图22-5　使用视图组件返回未编码的HTML片段

22.3.4　获取上下文数据

有关当前请求和父视图的详细信息会通过ViewComponentContext类的属性提供给视图组件，表22-5描述了该类提供的一些有用的属性。

表22-5　**ViewComponentContext**类定义的属性

属性	描述
Arguments	返回视图提供的参数字典，也可以通过Invoke方法接收
HtmlEncoder	返回一个可用于安全地编码HTML片段的HtmlEncoder对象
ViewComponentDescriptor	返回一个ViewComponentDescriptor对象，以提供视图组件的描述
ViewContext	从父视图返回ViewContext对象
ViewData	返回一个ViewDataDictionary对象，以提供对视图组件的视图数据的访问

ViewComponent基类提供了一组便于访问特定

上下文信息的属性，如表22-6所示。

表**22-6　ViewComponent**基类定义的属性

属性	描述
ViewComponentContext	返回一个ViewComponentContext对象
HttpContext	返回一个描述当前请求和正在准备的响应的HttpContext对象
Request	返回一个描述当前HTTP请求的HttpRequest对象
User	返回一个描述当前用户的IPrincipal对象
RouteData	返回一个描述当前请求的路由数据的RouteData对象
ViewBag	返回动态的ViewBag对象，可用于在视图组件和视图之间传递数据
ModelState	返回一个ModelStateDictionary对象，以提供模型绑定过程的详细信息
ViewContext	返回一个提供给父视图的ViewContext对象
ViewData	返回一个ViewDataDictionary对象，以提供对视图组件的视图数据的访问

Url	返回一个可用于生成URL的IUrlHelper对象

上下文数据可以各种方式用于帮助视图组件执行工作，包括改变数据的选择方式以及呈现不同的内容或视图。代码清单22-22使用路由数据来缩小City对象的选择范围。

代码清单22-22　在Components文件夹下的CitySummary.cs文件中使用上下文数据

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
        private ICityRepository repository;

        public CitySummary(ICityRepository repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            string target = RouteData.Values["id"] as s
tring;
```

```
                var cities = repository.Cities
                    .Where(city => target == null ||
                        string.Compare(city.Country, target
, true) == 0);
                return View(new CityViewModel{
                    Cities = cities.Count(),
                    Population = cities.Sum(c => c.Populati
on)
                });
            }
        }
}
```

浏览器使用路由中的id片段来指定国家数据，使用LINQ来过滤存储库中的对象。如果启动应用程序并请求默认URL，就会显示所有城市。可以通过请求URL（比如/Home/Index/USA）来缩小选择范围，这样可以将选择范围缩小到美国的城市，如图22-6所示。

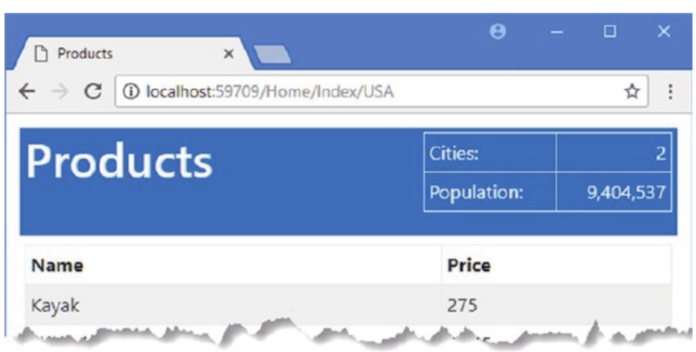


图22-6　在视图组件中使用上下文数据

使用参数从父视图提供上下文

父视图可以提供附加的上下文数据作为@await Component.Invoke表达式的参数。此功能可用于从父视图模型提供数据，或指导视图组件生成有关类型的内容。为了演示此功能，在Views/Home/Component/ CitySummary文件夹中创建一个名为CityList.cshtml的视图文件，内容如代码清单22-23所示。

代码清单22-23　Views/Home/Component/CitySummary文件夹下的CityList.cshtml文件的内容

```
@model IEnumerable<City>

<table class="table table-sm table-bordered">
    @foreach (var city in Model) {
        <tr>
            <td>@city.Name</td>
            <td class="text-right">
                @city.Population.ToString("#,###")
            </td>
```

```
        </tr>
    }
    <tr>
        <th>Total:</th>
        <td class="text-right">
            @Model.Sum(p => p.Population).ToString("#,#
##")
        </td>
    </tr>
</table>
```

添加第二个视图以允许在视图之间选择视图组件，根据添加到Invoke方法的参数完成该操作，如代码清单22-24所示。

代码清单22-24　在Components文件夹下的CitySummary.cs文件中选择视图

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace UsingViewComponents.Components {

    public class CitySummary : ViewComponent {
        private ICityRepository repository;

        public CitySummary(ICityRepository repo) {
```

```
            repository = repo;
        }
        public IViewComponentResult Invoke(bool showLis
t) {
            if (showList) {
                return View("CityList", repository.Citi
es);
            } else {
                return View(new CityViewModel {
                    Cities = repository.Cities.Count(),
                    Population = repository.Cities.Sum(
c => c.Population)
                });
            }
        }
    }
}
```

如果Invoke方法的showList参数为true，视图组件就选择CityList，并将存储库中的所有City对象作为视图模型传递。如果showList参数为false，就选择默认视图，并为视图模型提供一个CitySummary对象。

最后一步是在父视图中应用视图组件时提供上下文数据，方法是将匿名对象传递给Invoke方法，

如代码清单22-25所示。

代码清单22-25　在Views/Shared文件夹下的_Layout.cshtml文件中提供上下文数据

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                @await Component.InvokeAsync("CitySumma
ry", new { showList = true })
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，视图组件将接收父视图指定的值，并相应地进行响应，结果如图22-7所示。

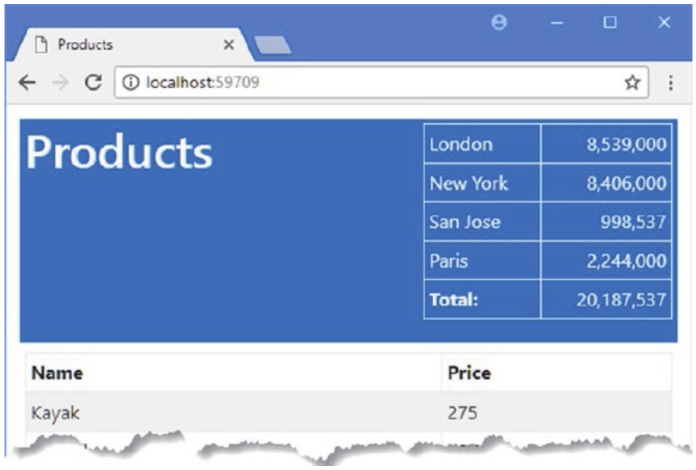


图22-7　向视图组件提供上下文数据

测试视图组件

视图组件遵循一般的MVC方法，从格式化和呈现模型数据的视图标记中分离出选择模型数据并进行处理的逻辑，从而可以轻松地执行单元测试。以下是示例应用程序中用于CitySummary的单元测试：

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
```

```
using UsingViewComponents.Models;
using UsingViewComponents.Components;
using Xunit;

namespace UsingViewComponents.Tests {
    public class SummaryViewComponentTests {

        [Fact]
        public void TestSummary() {

            // Arrange
            var mockRepository = new Mock<ICityReposito
ry>();
            mockRepository.SetupGet(m => m.Cities).Retu
rns(new List<City> {
                new City { Population = 100 },
                new City { Population = 20000 },
                new City { Population = 1000000 },
                new City { Population = 500000 }
            });
            var viewComponent
                = new CitySummary(mockRepository.Object
);

            // Act
            ViewViewComponentResult result
                = viewComponent.Invoke(false) as ViewVi
ewComponentResult;

            // Assert
            Assert.IsType(typeof(CityViewModel), result
.ViewData.Model);
            Assert.Equal(4, ((CityViewModel)result.View
Data.Model).Cities);
            Assert.Equal(1520100,
                ((CityViewModel)result.ViewData.Model).
```

```
Population);
        }
    }
}
```

为了安排测试，创建假的存储库并传递给CitySummary类的构造函数，以创建视图组件的新实例。对于测试的act部分，调用Invoke方法以提供结果对象。视图组件选择了Razor视图，因此将结果转换为ViewViewComponentResult，并通过提供的ViewData.Model属性访问视图模型对象。对于测试的Assert部分，检查视图模型数据的类型及其包含的值。

22.3.5　创建异步视图组件

本章迄今为止的所有示例使用的都是同步视图组件，可以识别它们，因为它们定义了Invoke方

法。如果视图组件依赖于异步API，则可以通过定义返回任务的InvokeAsync方法来创建异步视图组件。当Razor从InvokeAsync方法接收到任务时，将等待任务完成，然后将结果插入主视图。为了准备此例，在Solution Explorer窗格中右击UsingViewComponents项目，在弹出的菜单中选择Edit UsingViewComponents.csproj，并进行代码清单22-26所示的更改，以向项目添加包。

代码清单22-26　在UsingViewComponents.csproj文件中添加包

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0 " />
    <PackageReference Include="System.Net.Http" Version
="4.3.2" />
```

```
  </ItemGroup>

</Project>
```

System.Net.Http包提供了一个用于进行异步HTTP请求的API，这里将用它查询Apress网站。代码清单22-27显示了名为PageSize.cs的类文件的内容，将这个类文件添加到Components文件夹中，用于创建异步视图组件。

代码清单22-27　Components文件夹下的PageSize.cs文件的内容

```
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace UsingViewComponents.Components {

    public class PageSize : ViewComponent {

        public async Task<IViewComponentResult> InvokeA
sync() {
            HttpClient client = new HttpClient();
            HttpResponseMessage response
                = await client.GetAsync("http://apress.
com");
            return View(response.Content.Headers.Conten
```

```
tLength);
        }
    }
}
```

InvokeAsync方法通过第4章描述的async和await关键字来使用HttpClient类提供的异步API，并通过向Apress.com发送GET请求来获取所返回内容的长度。将长度传递给View方法，选择与视图组件关联的默认分部视图。

为了创建视图，在项目中添加Views/Shared/Components/PageSize文件夹，并在其中添加一个名为Default.cshtml的视图文件，内容如代码清单22-28所示。

代码清单22-28　Views/Shared/Components/PageSize文件夹下的Default.cshtml文件的内容

```
@model long
<div class="m-1 p-1 bg-info text-white">Page size: @Mod
el</div>
```

最后一步是在_Layout.cshtml文件中使用异步视图组件，如代码清单22-29所示。

代码清单22-29　在Views/Shared文件夹下的_Layout.cshtml文件中使用异步视图组件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                @await Component.InvokeAsync("CitySumma
ry",
                    new { showList = true })
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
    @await Component.InvokeAsync("PageSize")
</body>
</html>
```

如果启动应用程序，你将在浏览器中看到新增的内容，如图22-8所示。当运行示例时，显示的数字可能会更改，因为Apress经常更新其网站。

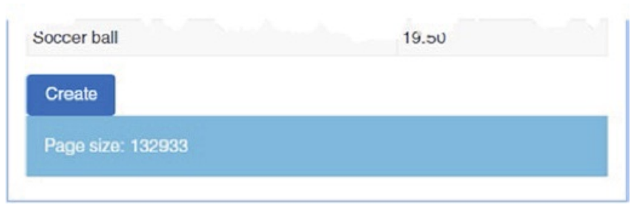


图22-8　创建异步视图组件

22.4　创建混合的控制器/视图组件类

视图组件通常提供由控制器深入处理的功能的摘要或快照。例如，对于购物车视图组件，通常会存在一个链接，目标是一个控制器，该控制器提供购物车中商品的详细列表，并且可以用于筛选并完成购买。

在这种情况下，可以创建混合的控制器/视图组件类，这样可以将相关功能组合在一起并减少代

码重复。为了演示，将一个名为CityController.cs的类文件添加到Controllers文件夹中，并用它定义代码清单22-30所示的控制器。

代码清单22-30　Controllers文件夹下的CityController.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using UsingViewComponents.Models;

namespace UsingViewComponents.Controllers {

    [ViewComponent(Name = "ComboComponent")]
    public class CityController : Controller {
        private ICityRepository repository;

        public CityController(ICityRepository repo) {
            repository = repo;
        }
        public ViewResult Create() => View();

        [HttpPost]
        public IActionResult Create(City newCity) {
            repository.AddCity(newCity);
            return RedirectToAction("Index", "Home");
        }
```

```
        public IViewComponentResult Invoke() => new ViewViewComponentResult() {
            ViewData = new ViewDataDictionary<IEnumerable<City>>(ViewData,
                repository.Cities)
        };
    }
}
```

将ViewComponent特性应用于不从ViewComponent基类继承且名称不以ViewComponent结尾的类，这意味着正常的发现过程通常不会将类看作视图组件。Name属性用于设置在父视图中使用@Component.Invoke表达式应用类时可以引用类的名称。在此例中，使用Name属性将类的视图组件部分的名称设置成ComboComponent。该名称将用于调用视图组件并查找视图。

因为混合类不能从ViewComponent基类继承，所以它们无法访问IViewComponentResult对象提供的便利方法，这意味着必须直接创建

ViewViewComponentResult对象，就像POCO视图组件中所需要的那样。

22.4.1　创建混合视图

混合类需要两组视图——当类用作控制器时渲染的视图以及在将类用作视图组件时呈现的视图。首先，创建Views/City文件夹，并添加一个名为Create.cshtml的视图文件，内容如代码清单22-31所示。

代码清单22-31　Views/City文件夹下的Create.cshtml文件的内容

```
@model City
@{
    ViewData["Title"] = "Create City";
    Layout = "_Layout";
}

<form method="post" asp-action="Create">
    <div class="form-group">
        <label asp-for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
```

```
    <div class="form-group">
        <label asp-for="Country">Country:</label>
        <input class="form-control" asp-for="Country" /
>
    </div>
    <div class="form-group">
        <label asp-for="Population">Population:</label>
        <input class="form-control" asp-for="Population
" />
    </div>
    <button type="submit" class="btn btn-primary">Creat
e</button>
    <a class="btn btn-secondary" asp-controller="Home"
       asp-action="Index">
        Cancel
    </a>
</form>
```

Create视图提供了一个简单的用于创建City对象的表单。Create按钮会向City控制器的Create操作方法发送POST请求，而Cancel按钮会向Home控制器的Index操作方法发送GET请求。

接下来，创建Views/Shared/Components/ComboComponent文件夹，并添加一个名为Default.cshtml的视图文件，内

容如代码清单22-32所示。将分部视图放置在Views/Shared文件夹中，视图组件的名称将包含在用于定位视图的路径中。

代码清单22-32　Views/Shared/Components/ComboComponent文件夹下的Default.cshtml文件的内容

```
@model IEnumerable<City>

<table class="table table-sm table-bordered">
    <tr>
        <td>Biggest City:</td>
        <td>
            @Model.OrderByDescending(c => c.Population)
.First().Name
        </td>
    </tr>
</table>
<a class="btn btn-sm btn-info" asp-controller="City" as
p-action="Create">
    Create City
</a>
```

这个分部视图接收使用LINQ排序的City对象序列，以选择具有最大Population值的那个。还有一个可设置样式为按钮的锚点元素，按钮将指向City

控制器的Create操作方法。

提　示

代码清单22-32明确指定了用于a元素的City控制器。可使用父视图提供的上下文数据生成URL，这意味着默认控制器就是处理请求的控制器，而不是使用视图组件的控制器。如果省略asp-controller属性，链接将在Home控制器上定位Create操作方法。

22.4.2　应用混合类

最后一步是使用ViewComponent特性指定的名称将混合类作为视图组件应用于共享布局，如代码清单22-33所示。

代码清单22-33　在Views/Shared文件夹下的_Layout.cshtml文件中应用混合类

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.mi
n.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div class="bg-primary m-1 p-1">
        <div class="row text-white">
            <div class="col-7"><h1>Products</h1></div>
            <div class="col-5">
                @await Component.InvokeAsync("ComboComp
onent")
            </div>
        </div>
    </div>
    <div class="m-1 p-1">@RenderBody()</div>
    @await Component.InvokeAsync("PageSize")
</body>
</html>
```

结果是由自己的集成控制器（或者如果喜欢，也可以是具有自己的集成视图组件的控制器）备份的视图组件。如果运行应用程序，你将看到伦敦被列为人口最多的城市。单击Create按钮，你将看到一个表单，可以向应用程序添加新的城市。填写并提交表单，控制器将收到数据，更新存储库，并将浏览器重定向到应用程序的默认URL。如果添加人口更多的城市，视图组件的输出结果将会改变，如图22-9所示。

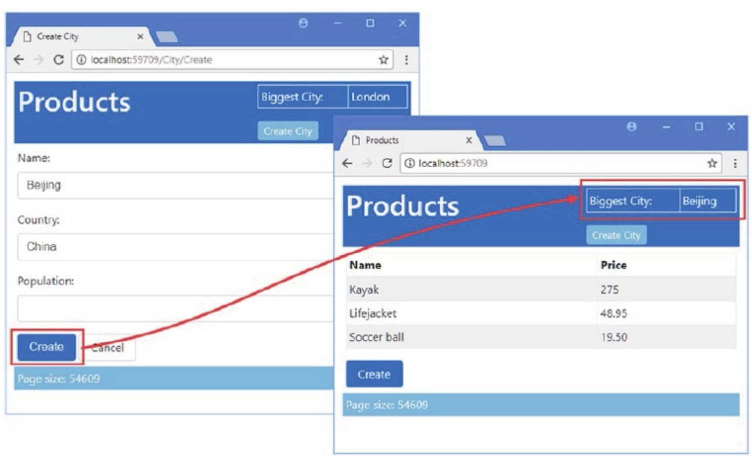


图22-9　使用混合的控制器/视图组件类

22.5　小结

本章介绍了视图组件，这是ASP.NET Core MVC的新增功能，以替代以前的MVC版本的子操作功能。本章演示了如何创建POCO视图组件以及如何使用ViewComponent基类，还展示了组件可以生成的3种不同类型的结果，包括在父视图中包含分部视图的选择。本章最后演示了如何将视图组件功能添加到控制器类中，以减少代码重复并简化应用程序。下一章将介绍标签助手，用于在视图中转换HTML元素。

第23章　标签助手

标签助手是ASP.NET Core MVC中引入的新特性，用于在视图中使用C#类转换HTML元素。通常的用法包括使用应用程序的路由配置来为表单生成URL、确保特定类型的元素样式能够一致地被修饰、将自定义的缩写元素替换为常用的内容片段等。本章将介绍标签助手是如何工作的，以及定制的标签助手是如何创建和应用的。第24章将说明内置的支持HTML表单的标签助手，第25章将说明MVC提供的其他内置标签助手。表23-1介绍了标签助手的背景。

标签助手组件

ASP.NET Core 2平台引入了标签助手，从而可以在发送到客户端的响应中修改特定的HTML片段，它们也可以通过标签助手来使用。因为它难以使用，对于大多数MVC开发人员来说用处不大，所以这里不再介绍。如果要转换发送到客户端的响应中的内容，请使用本章介绍的标签助手特性。

表**23-1**　标签助手的背景

问题	答案
标签助手是什么？	标签助手是操作HTML元素的类，可使用某种方式改变它们，提供附加的内容，也可使用新的内容整个替换它们
标签助手有何作用？	标签助手允许使用C#逻辑来呈现或转换视图内容，确保发送到客户端的HTML反映应用程序的状态
如何使用标签助手？	可基于元素的类名或者使用HTMLTargetElement特性来选择应用标签助手的HTML元素。当视图被呈现时，元素通过标签助手被转换并包含在发送到客户端的HTML中
标签助手有	

何缺陷或限制？	标签助手很容易被过度使用，用于生成复杂的HTML内容片段，使用视图组件更易于做到
是否有其他的替代方案？	不一定必须使用标签助手，但是它们可以使你在MVC应用程序中更加容易地生成复杂的HTML内容

表23-2列出了本章要介绍的操作。

表**23-2**　本章要介绍的操作

操作	方法	代码清单
转换HTML元素	创建标签助手，然后使用@addTagHelper表达式在视图中或在视图导入文件中进行注册	代码清单23-10～代码清单23-12
管理标签助手的作用范围	使用HtmlTargetElement特性	代码清单23-13～代码清单23-17
使用缩写元素	使用TagHelperOutput对象生成替换元素	代码清单23-18和代码清单23-19
为目标元素插入内容或环绕内容	使用TagHelperOutput提供的Pre-和Post-属性	代码清单23-22和代码清单23-23
在标签助手中接收上下文数据	使用ViewContext和HtmlAttributeNotBound特性装饰属性	代码清单23-24和代码清单23-25

访问视图模型	使用ModelExpression属性	代码清单23-26和代码清单23-27
组织标签助手	使用TagHelperContext.Items属性	代码清单23-28和代码清单23-29
抑制元素	使用SuppressOutput方法	代码清单23-30和代码清单23-31

23.1 准备示例项目

在本章中，使用ASP.NET Core Web Application（.NET Core）模板新建名为Cities的Empty项目。

23.1.1 创建模型和存储库

创建Models文件夹，添加名为City.cs的类文件，并用它定义代码清单23-1所示的类。

代码清单23-1 Models文件夹下的City.cs文件的内容

```
namespace Cities.Models {
```

```
    public class City {
        public string Name { get; set; }
        public string Country { get; set; }
        public int? Population { get; set; }
    }
}
```

为了给City对象创建存储库，在Models文件夹中添加名为Repository.cs的类文件，并用它定义代码清单23-2所示的接口和实现。

代码清单23-2　Models文件夹下的Repository.cs文件的内容

```
using System.Collections.Generic;

namespace Cities.Models {

    public interface IRepository {

        IEnumerable<City> Cities { get; }
        void AddCity(City newCity);
    }

    public class MemoryRepository : IRepository {

        private List<City> cities = new List<City> {
            new City { Name = "London", Country = "UK",
 Population = 8539000},
            new City { Name = "New York", Country = "US
A", Population = 8406000 },
```

```
            new City { Name = "San Jose", Country = "US
A", Population = 998537 },
            new City { Name = "Paris", Country = "Franc
e", Population = 2244000 }
        };

        public IEnumerable<City> Cities => cities;

        public void AddCity(City newCity) {
            cities.Add(newCity);
        }
    }
}
```

23.1.2　创建控制器、布局与视图

本章的示例只需要一个控制器。创建Controllers文件夹，添加名为HomeController.cs的类，然后用它定义代码清单23-3所示的控制器。

代码清单23-3　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;

namespace Cities.Controllers {

    public class HomeController : Controller {
```

```
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }
        public ViewResult Index() => View(repository.Ci
ties);

        public ViewResult Create() => View();

        [HttpPost]
        public IActionResult Create(City city) {
            repository.AddCity(city);
            return RedirectToAction("Index");
        }
    }
}
```

控制器提供了Index操作方法用于列出存储库中的对象，一对Create方法将允许用户使用表单来创建新的City对象，可延续使用与前面章节中相同的模式。

应用程序中的视图将使用共享的布局。创建Views/Shared文件夹，在Views/Shared文件夹中添加名为_Layout.cshtml的布局文件，加入代码清单23-4

所示的标记内容。

注　意

本章的目的是演示标签助手如何工作，布局和示例应用程序中的视图仅仅使用标准的HTML元素来编写，它们将被介绍的其他标签助手替换。

代码清单23-4　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
```

```
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

然后，创建Views/Home文件夹，添加名为Index.cshtml的文件，内容如代码清单23-5所示。

代码清单23-5　Views/Home文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<City>

@{  Layout = "_Layout"; }

<table class="table table-sm table-bordered">
    <thead class="bg-primary text-white">
        <tr>
            <th>Name</th>
            <th>Country</th>
            <th class="text-right">Population</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var city in Model) {
            <tr>
                <td>@city.Name</td>
                <td>@city.Country</td>
```

```
                <td class="text-right">@city.Population
?.ToString("#,###")</td>
            </tr>
        }
    </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</
a>
```

Index视图使用City对象的序列来填充表格，还包含一个a元素用来指向URL地址/Home/Create，a元素已使用Bootstrap修饰为按钮样式。对于第二个视图，添加名为Create.cshtml的文件到Views/Home文件夹中，内容如代码清单23-6所示。

代码清单23-6　Views/Home文件夹下的Create.cshtml文件的内容

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
```

```
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" />
    </div>

    <button type="submit" class="btn btn-primary">Add</button>
    <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

这里还在Views文件夹中创建了名为_ViewImports.cshtml的视图导入文件，在其中添加代码清单23-7所示的表达式。这允许引用Models文件夹中的类而不需要使用命名空间。

代码清单23-7　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using Cities.Models
```

示例项目中的视图基于Bootstrap CSS包。为了

将BootStrap添加到示例项目中，使用Bower Configuration File模板在项目的根目录中创建名为bower.json的文件，并添加代码清单23-8所示的包到dependencies部分。

代码清单23-8　在Cities文件夹下的bower.json文件中添加Bootstrap

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

23.1.3　配置应用程序

最终的准备步骤是配置应用程序，如代码清单23-9所示。本书在所有示例项目中使用同样的基本配置，另外使用单例生命周期作为服务注册存储库。

代码清单23-9　Cities文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Cities.Models;

namespace Cities {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

如果运行应用程序，将会显示来自存储库默认

创建的City对象的清单。单击Create按钮，填写表单，然后单击Add按钮；新的对象将会被添加到存储库中，如图23-1所示。

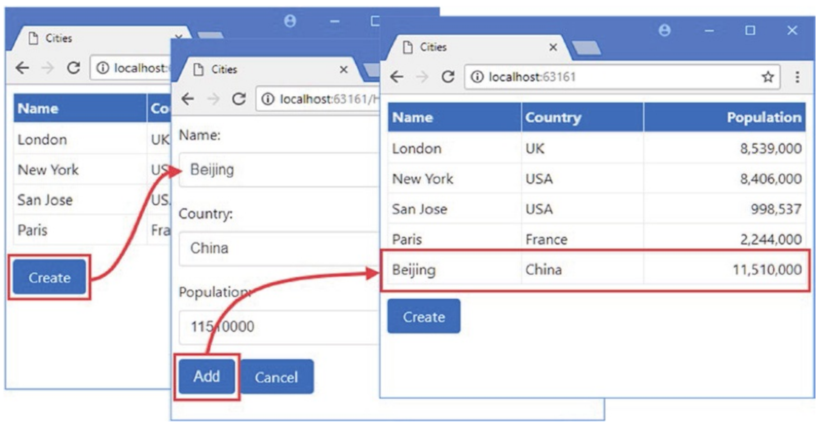


图23-1　运行应用程序

23.2　创建标签助手

与许多的MVC特性一样，理解标签助手的最佳方式就是创建它们，这可以揭示它们是如何运作以及融入应用程序的。后面将介绍创建和应用标签助手的过程，将为button元素应用Bootstrap CSS样式，以便对如下元素进行转换：

```
...
<button type="submit" bs-button-color="danger">Add</but
ton>
...
```

转换后的形式如下：

```
...
<button type="submit" class="btn btn-danger">Add</butto
n>
...
```

标签助手将会识别bs-button-color属性，然后使用该属性的值设置发送到浏览器的元素的class属性。这种转换虽然不是很有用，但可以为说明标签助手如何工作打下基础。

23.2.1 定义标签助手类

标签助手可以在项目的任何位置定义，但将它们保存在一起是有帮助的，与大多数MVC组件不同，它们在使用之前需要注册。在Infrastructure/TagHelpers文件夹中创建这些标签助

手，并将它们添加到项目中。

标签助手类需要派生自TagHelper类，且定义在命名空间Microsoft.AspNetCore.Razor.TagHelpers中。为了创建标签助手，在Infrastructure/TagHelpers文件夹中添加名为ButtonTagHelper.cs的文件，然后用它定义代码清单23-10所示的类。

代码清单23-10　Infrastructure/TagHelpers文件夹下的ButtonTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    public class ButtonTagHelper : TagHelper {

        public string BsButtonColor { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.Attributes.SetAttribute("class", $"b
```

```
tn btn-{BsButtonColor}");
        }
    }
}
```

TagHelper类定义了Process方法，该方法可由子类重写以实现转换元素的行为。标签助手的名称由被转换元素的名称加上TagHelper组成。在这个示例中，类名ButtonTagHelper告诉MVC这是用于转换button元素的标签助手。标签助手的作用域可以使用特性进行拓宽或限制，这里是默认行为。

提 示

异步标签助手可以通过重写ProcessAsync方法而不是Process方法来创建，但是对大多数标签助手是不必要的，而倾向于创建小的且专注于改变的

HTML元素。默认的ProcessAsync实现会调用Process方法。

---

1. 接收上下文数据

标签助手通过TagHelperContext类的实例来接收关于转换元素的信息，实例可作为Process方法的参数接收，且定义了表23-3定义的属性。

表**23-3 TagHelperContext**类定义的属性

属性	描述
AllAttributes	返回一个只读的待转换元素的特性字典，以属性名称作为索引
Items	返回一个用于协调标签助手的字典
UniqueId	返回待转换元素的唯一标识

尽管可以通过AllAttributes字典访问待转换元

素的属性详情，但更方便的方式是定义一个属性，属性名对应你感兴趣的元素属性，如下所示：

```
...
public string BsButtonColor { get; set; }
...
```

当标签助手被应用时，MVC检查标签助手类定义的属性，并使用名称得到完整匹配的HTML元素属性的值来设置。作为上述处理过程的一部分，MVC将试图转换HTML属性的值为C#属性的类型，bool类型的属性可以接收HTML属性的true或false值，int类型的属性可以接收类似于1和2的HTML属性值。

HTML Helper发生了什么？

早期版本的ASP.NET MVC使用HTML helper来

生成表单元素。HTML helper是一组通过Razor表达式访问的以@Html开始的方法，创建input元素并填充属性的方式可能如下所示：

```
...
@Html.TextBoxFor(m => m.Population)
...
```

HTML helper表达式的问题是它们不适合HTML元素的结构，这导致表达式十分笨拙，类似于下面这样：

```
...
@Html.TextBoxFor(m => m.Population, new { @class = "form-control" })
...
```

属性不得不表示为动态对象，并且如果名称为C#保留字，例如class，那么不得不使用@前缀。由于HTML元素需要变得越来越复杂，HTML helper表达式也变得更加笨拙。标签助手通过使用HTML属性避免了这一点，类似于下面这样：

```
...
<input class="form-control" asp-for="Population" />
...
```

结果就是生成更自然的HTML，并且处理的视图也更易读和易于理解。MVC仍然支持HTML helper（并且标签助手在幕后使用HTML helper），可以在原来开发的MVC5视图中继续使用HTML Helper来实现后向兼容性，但是，新的视图应当使用标签助手。

属性的名称会从默认的HTML风格执行自动转换，比如从bs-button-color转换为C#风格的BsButtonColor。可以使用除了asp-（由微软使用）和data（由发送到客户端的自定义属性保留）以外的任何前缀。本例的Process方法使用BsButtonColor属性来接收应用于button元素的颜色模式，如下所

示：

```
...
output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
...
```

没有关联HTML属性的属性值不会被设置，这意味着应当检查并确认没有在处理null或默认值。

提　示

使用HTML属性名称用于标签助手属性并不总是导致易读或易于理解的类。可以使用HtmlAttributeName特性来打破属性名称与其表示的属性之间的联系，从而用来指定属性将要表示的HTML属性。

2．产生输出

Process方法通过配置作为参数接收的TagHelperOutput对象来转换元素。TagHelperOuput从自身在Razor视图中出现的位置开始输出HTML元素，并通过表23-4所示的属性和方法来改变元素。

表**23-4　TagHelperOutput**类定义的属性和方法

名称	描述
TagName	该属性用于获取或设置输出元素的标记名称
Attributes	该属性返回用于输出元素的属性字典
Content	该属性返回TagHelperContent对象，用于设置元素内容
PreElement	该属性返回TagHelperContent对象，用于在视图中输出元素之前插入内容
PostElement	该属性返回TagHelperContent对象，用于在视图中输出元素之后追加内容

PreContent	该属性返回TagHelperContext对象，用于在输出元素的内容之前插入内容
PostContent	该属性返回TagHelperContext对象，用于在输出元素的内容之后插入内容
TagMode	该属性指定输出元素的输出编写方式，可使用来自TagMode枚举的值
SupressOuput()	该方法用于防止元素被包含到视图中

在ButtonTagHelper类中，可使用Attributes字典添加class属性到HTML元素中，用于设置Bootstrap按钮样式，包含来自BsButtonColor属性的值，这意味着可以使用Bootstrap名称（比如primary、info、和danger）来指定不同的颜色。

23.2.2　注册标签助手

标签助手类只有在使用Razor的@addTagHelper表达式注册之后才能使用。被应用的视图集将基于@addTagHelper表达式的应用位置。对于单个视图，表达式出现在视图本身之中。对于应用程序中

的一组视图，表达式出现在包含这些视图的文件夹或父文件夹中名为_ViewImports.cshtml的文件中。比如，/Views/Home/_ViewImports.cshtml文件中使用@addTagHelper表达式注册的标签助手将作用于Home控制器的所有视图。为了让本章创建的标签助手可以被整个应用程序中的视图使用，在Views/_ViewImports.cshtml文件添加@addTagHelper表达式，如代码清单23-11所示。

代码清单23-11　在Views文件夹下的_ViewImports.cshtml文件中注册标签助手

```
@using Cities.Models
@addTagHelper Cities.Infrastructure.TagHelpers.*, Cities
```

参数的第一部分指定了标签助手的类名，支持通配符；第二部分指定了定义所在的程序集名称。以上代码注册了Cities程序集中命名空间Cities.Infrastructure.TagHelpers里的所有标签助手。

23.2.3　使用标签助手

最终的步骤是使用标签助手转换元素。代码清单23-12从Create.cshtml视图文件的button元素中删除了class属性，替换为ButtonTagHelper类中定义的属性。

代码清单23-12　在Views/Home文件夹下的Create.cshtml视图文件中使用标签助手

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>
```

```
    <button type="submit" bs-button-color="danger">Add</button>
    <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

如果运行应用程序，单击Create按钮，浏览器将会访问/Home/Create，你将会看到Add按钮的颜色和样式发生了变化，如图23-2所示。



图23-2　使用标签助手修饰按钮

测试标签助手

用于标签助手的单元测试是一个相对简单的过程。下面是为代码清单23-12所示的标签助手提供

的示例测试:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Cities.Infrastructure.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Xunit;

namespace Cities.Tests {

    public class TagHelperTests {

        [Fact]
        public void TestTagHelper() {
            // Arrange
            var context = new TagHelperContext(
                new TagHelperAttributeList(),
                new Dictionary<object, object>(),
                "myuniqueid");

            var output = new TagHelperOutput("button",
                new TagHelperAttributeList(), (cache, e
ncoder) =>
                    Task.FromResult<TagHelperContent>
                        (new DefaultTagHelperContent())
);

            // Act
            var tagHelper = new ButtonTagHelper {
                BsButtonColor = "testValue"
            };
            tagHelper.Process(context, output);
```

```
            // Assert
            Assert.Equal($"btn btn-{tagHelper.BsButtonC
olor}",
                output.Attributes["class"].Value);
        }
    }
}
```

在以上单元测试中，多数工作是设置TagHelperContext和TagHelperOutput对象，以便它们可以被传递给标签助手的Process方法，以检查并确保HTML元素被正确转换。准备用于测试的标签助手所需的工作量自然取决于它们所操作的HTML的复杂性以及转换程度。但是，大多数标签助手相对简单，可以按照前面讲述的基本模式进行测试。

23.2.4 管理标签助手的作用域

标签助手将被应用于给定类型的所有元素，这意味着之前创建的ButtonTagHelper类的Process方

法，将由应用程序中每个视图的每个button元素调用。这不总是有用的。为了针对这个问题提供示例，在Create.cshtml文件中添加另一个button元素，如代码清单23-13所示。

代码清单23-13　在Views/Home文件夹下的Create.cshtml文件中添加另一个button元素

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>

    <button type="submit" bs-button-color="danger">Add<
/button>
```

```
    <button type="reset" class="btn btn-primary" >Reset
</button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

新的 button 元素已经拥有 class 属性，并且不需要通过ButtonTagHelper类执行转换。但是，如果运行应用程序并导航到URL地址/Home/Create，你将会看到有问题产生，如图23-3所示。

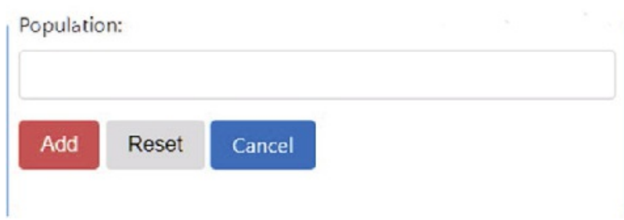


图23-3　标签助手的默认作用域的效果

可以通过查看发送到浏览器的HTML来查看格式缺失的原因，这揭示了class属性的问题，如下所示：

```
<button type="reset" class="btn btn-">Reset</button>
```

MVC应用ButtonTagHelper到新的button元素，但是由于HTML元素没有相应的bs-button-color属性，因此没有设置BsButtonColor属性的值。结果，标签助手使用不正确的Bootstrap样式替换class属性，生成格式缺失的元素。

1．限制标签助手的作用域

解决这个问题有两种方式。一种方式是修改ButtonTagHelper类，以使其对可能遇到的button元素敏感。对于示例应用程序来说，这需要增加额外的检查，例如确认是否拥有bs-button-color属性，并确保在定义了class属性之后不会被替换掉。这种方式的问题在于，随着包含button元素的视图不断添加到应用程序中，标签助手类会变得越来越复杂，并且基于ButtonTagHelper的所有新添加的额外复杂条件描述并不执行转换。

另一种方式是允许标签助手描述对使用方式的限制，缩小作用域。可使用HtmlTargetElement特性限制标签助手的作用域，如代码清单23-14所示。

代码清单23-14　限制Infrastructure/TagHelpers文件夹下的ButtonTagHelper.cs的作用域

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("button", Attributes = "bs-button-color", ParentTag = "form")]
    public class ButtonTagHelper : TagHelper {

        public string BsButtonColor { get; set; }

        public override void Process(TagHelperContext context,
                                     TagHelperOutput output) {

            output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
        }
    }
}
```

HtmlTargetElement特性描述了标签助手应用的

元素。第一个参数指定了元素类型，并支持表23-5所示的额外属性。

表**23-5　HtmlTargetElement**特性支持的属性

属性	描述
Attributes	用于指定标签助手应作用于拥有给定属性集的指定元素，可使用以逗号分隔属性的列表来提供。元素必须拥有全部指定的属性。以星号结束的属性名称将被视为前缀，所以bs-button-*将匹配bs-button-color、bs-button-size等
ParentTag	用于指定标签助手仅应作用于包含指定类型元素的元素
TagStructure	用于指定标签助手仅应作用于元素标记结构对应于TagStructure枚举中给定值的元素，其中定义了Unspecified、NormalOrSelfClosing和WithoutEndTag

代码清单23-14限制了ButtonTagHelper类仅作用于父元素是form元素，且拥有bs-button-color属性的button元素。如果运行应用程序并导航到/Home/Create，你将看到 Reset 按钮因为缺失要求的属性而不再被转换，如图 23-4所示。

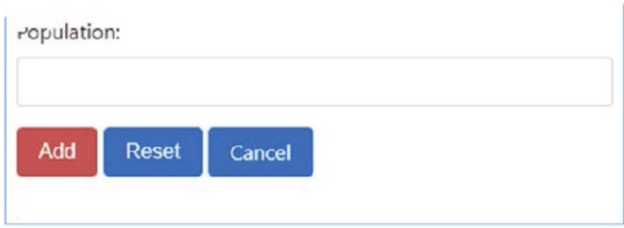


图23-4　限制标签助手的作用域

2．拓宽标签助手的作用域

特性HtmlTargetElement也可以用来拓宽标签助手的作用域，以便匹配更广泛的元素。当需要对多种不同类型的元素执行相同的转换时，这是很有用的。这与基于标签助手类名匹配元素的前提相违背，如代码清单23-15所示。

代码清单23-15　在Infrastructure/TagHelpers文件夹下的ButtonTagHelper.cs中拓宽标签助手的作用域

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement(Attributes = "bs-button-color",
ParentTag = "form")]
    public class ButtonTagHelper : TagHelper {
```

```
        public string BsButtonColor { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.Attributes.SetAttribute("class", $"b
tn btn-{BsButtonColor}");
        }
    }
}
```

以上代码忽略了HtmlTargetElement中的元素类型，这意味着标签助手将被应用于任何拥有bs-button- color属性的元素，而无论元素的类型如何。代码清单23-16修改了表单中的a元素，使用与按钮元素相同的Bootstrap样式集，以便可以由标签助手进行转换。

代码清单23-16　在Views/Home文件夹下的Create.cshtml文件中修改锚元素

```
@model City

@{  Layout = "_Layout"; }
```

```
<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>

    <button type="submit" bs-button-color="danger">Add<
/button>
    <button type="reset" class="btn btn-primary" >Reset
</button>
    <a bs-button-color="primary" href="/Home/Index">Can
cel</a>
</form>
```

标签助手的作用域更广泛意味着对于不同类型的元素不必重复同样的操作。但是注意，应用程序中的视图内容在演进的时候，很容易创建匹配过于宽泛的元素的标签助手。更平衡的方式是应用多次HtmlTargetElement特性，以组合限制定义匹配的方

式指定元素的完整集合，如代码清单23-17所示。

代码清单23-17　在Infrastructure/TagHelpers文件夹下的ButtonTagHelper.cs文件中平衡标签助手的作用域

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("button", Attributes = "bs-butto
n-color", ParentTag = "form")]
    [HtmlTargetElement("a", Attributes = "bs-button-col
or", ParentTag = "form")]
    public class ButtonTagHelper : TagHelper {

        public string BsButtonColor { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.Attributes.SetAttribute("class", $"b
tn btn-{BsButtonColor}");
        }
    }
}
```

上述配置对应用程序有着同样的效果，但请确保在未来的开发过程中，出于其他的原因添加bs-

button-color属性到其他元素时不会导致问题。

排序标签助手的执行

作为一般原则，好的做法是对于特定的HTML元素最好仅仅应用一个标签助手，这是因为很容易导致一个标签助手破坏另一个标签助手的情况。如果需要应用多个标签助手，那么通过设置Order属性可以控制执行的顺序。管理执行顺序可以帮助你在标签助手之间最小化冲突，虽然仍然容易导致问题。

23.3　高级标签助手特性

前面演示了如何创建基本的标签助手，但是仅

仅展示了表面的可能性，后面将展示标签助手提供的更高级的用法和特性。

23.3.1　创建缩写元素

标签助手并不限于转换标准HTML元素，也可以用来转换常用内容的自定义元素。这是一个使得创建更简洁、意图更明显的视图的有用特性。为了演示，使用自定义元素替换Create.cshtml文件中的button元素，如代码清单23-18所示。

代码清单23-18　在Create.cshtml文件中添加自定义元素

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
```

```
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>
    <formbutton type="submit" bg-color="danger" />
    <formbutton type="reset" />
    <a bs-button-color="primary" href="/Home/Index">Can
cel</a>
</form>
```

formbutton元素不是HTML规范的一部分，也不被浏览器所理解。相反，使用这些元素作为生成表单所需的button元素的缩写。在Infrastructure/TagHelper文件夹中添加一个名为FormButtonTagHelper.cs的类文件，并用它定义代码清单23-19所示的类。

代码清单23-19　Infrastructure/TagHelpers文件夹下的FormButtonTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("formbutton")]
```

```
    public class FormButtonTagHelper : TagHelper {

        public string Type { get; set; } = "Submit";

        public string BgColor { get; set; } = "primary"
;

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.TagName = "button";
            output.TagMode = TagMode.StartTagAndEndTag;
            output.Attributes.SetAttribute("class", $"b
tn btn-{BgColor}");
            output.Attributes.SetAttribute("type", Type
);
            output.Content.SetContent(Type == "submit"
? "Add" : "Reset");
        }
    }
}
```

提　示

当处理不是HTML规范一部分的自定义元素时，必须应用HtmlTargetElement特性，并指定元素名称，如代码清单23-19所示。基于类型将标签助手应用到元素的便捷方式仅仅适用于标准元素名称。

Process方法使用TagHelperOuput对象的属性来生成复杂的不同元素：TagName属性用于指定button元素，TagMode属性用于指定元素使用开始和结束标记写入。Attributes.SetAttribute方法用于定义Bootstrap样式的class属性，Content属性用于设置元素内容。

提　示

代码清单23-19设置了输出元素的type属性。这在输出元素中忽略了标签助手定义的任何属性，这通常是个好主意，因为阻止了用于标签助手的属性出现在发送给浏览器的HTML中。但是，在这种情况下需要使用type属性来配置标签助手，以便它们出现在输出元素中。

设置TagName属性是很重要的，因为对于自定义元素，输出元素默认使用同样的样式。代码清单23-19使用了自结束标记：

```
...
<formbutton type="submit" bg-color="danger" />
...
```

为了在输出元素中包含内容，不得不显式指定

TagMode.StartTagAndEndTag枚举值，以便分离开始和结束标记。

Content属性会返回一个TagHelperContent实例，用来设置元素的内容。表23-6列出了TagHelperContent定义的主要方法。

表**23-6　TagHelperContent**定义的主要方法

方法	描述
SetContent(text)	设置输出元素的内容，字符串参数将被编码以便被HTML元素安全包含
SetHtmlContent(html)	设置输出元素的内容，字符串参数假定已经安全编码，要慎用
Append(text)	安全编码指定的字符串，并添加到输出元素的内容中
AppendHtml(html)	添加指定的字符串到输出元素的内容中，而不执行任何编码，要慎用
Clear()	删除输出元素的内容

在代码清单23-19中，标签助手基于type特性的值，使用SetContent方法来设置输出元素的内容，值由Type属性提供。如果运行应用程序，并访问/Home/Create，你将会看到自定义的formbutton元素将被替换为标准的HTML元素，所以需要转换以下元素：

```
...
<formbutton type="submit" bg-color="danger" />
<formbutton type="reset" />
...
```

转换后的元素如下：

```
<button class="btn btn-danger" type="submit">Add</button>
<button class="btn btn-primary" type="reset">Reset</button>
```

23.3.2　前置和追加内容与元素

TagHelperOutput类提供了四个属性，使得更容易在视图中注入新的内容，以便环绕元素或内容，

如表23-7所示。后面将说明如何在目标元素中插入内容。

表**23-7**　用于添加内容与元素的**TagHelperOutput**属性

属性	描述
PreElement	用于在目标元素之前插入元素
PostElement	用于在目标元素之后插入元素
PreContent	用于在目标元素的任何现存内容之前插入内容
PostContent	用于在目标元素的任何现存内容之后插入内容

1．在环绕的输出元素中插入内容

第一组TagHelperOuput属性是PreElement和PostElement，它们用来在输出元素之前和之后插入元素到视图中。作为演示，添加名为ContentWrapperTagHelper.cs的类文件，并用它创建代码清单23-20所示的标签助手类。

代码清单23-20　Infrastructure/TagHelpers文件夹下的ContentWrapperTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("div", Attributes = "title")]
    public class ContentWrapperTagHelper : TagHelper {

        public bool IncludeHeader { get; set; } = true;
        public bool IncludeFooter { get; set; } = true;

        public string Title { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.Attributes.SetAttribute("class", "m-
1 p-1");

            TagBuilder title = new TagBuilder("h1");
            title.InnerHtml.Append(Title);

            TagBuilder container = new TagBuilder("div"
);
            container.Attributes["class"] = "bg-info m-
1 p-1";

            container.InnerHtml.AppendHtml(title);
```

```
            if (IncludeHeader) {
                output.PreElement.SetHtmlContent(contai
ner);
            }

            if (IncludeFooter) {
                output.PostElement.SetHtmlContent(conta
iner);
            }
        }
    }
}
```

该标签助手用于转换拥有title属性的div元素，该div元素使用PreElement和PostElement属性来添加页头和页脚元素以环绕输出元素。

当生成新的HTML元素时，可以使用标准的C#字符串格式来创建输出内容，但除非用于最简单的元素，否则这是一种笨拙且易错的方式。更稳健的方式是使用TagBuilder类，它定义在命名空间Microsoft. AspNetCore.Mvc.Rendering 中，该类支持以更结构化的方式创建元素。TagHelperContent 类定义了接收TagBuilder对象的方法，使得在标签助

手中更易于创建HTML内容。

这个标签助手使用TagBuilder类来创建h1元素，其包含在使用Bootstrap类修饰的div元素内。可选的bool类型的include-header与include-footer属性用于指定内容插入何处，默认行为是在输出元素之前和之后添加元素。代码清单23-21更新了共享布局，以便包含一个将被标签助手转换的元素。

代码清单23-21　在Views/Shared文件夹下的_Layout.cshtml文件中使用标签助手

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div title="Cities">@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，你将会看标签助手被应用到应用程序中，还在所有页面中添加了页头和页脚，如图23-5所示。

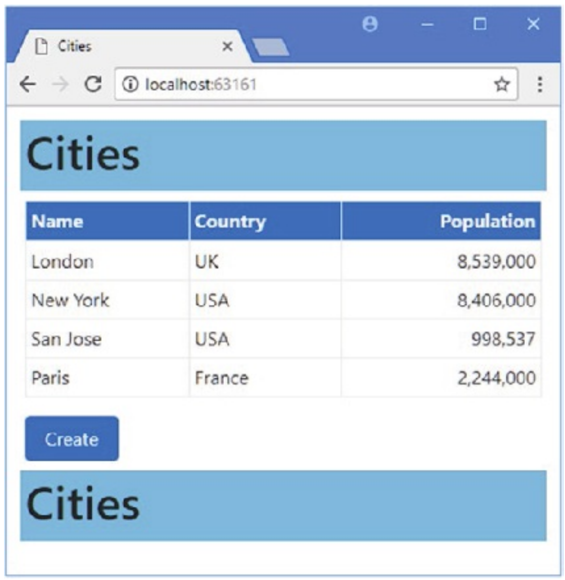


图23-5　使用标签助手插入HTML元素

2．在输出元素中插入内容

PreContent和PostContent属性用于在输出元素内环绕源内容插入内容。为了演示，在Infrastructure/TagHelpers文件夹中添加名为TableCellTagHelper.cs的类文件，并用它定义代码

清单23-22所示的类。

代码清单23-22　Infrastructure/TagHelpers文件夹下的TableCellTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("td", Attributes = "wrap")]
    public class TableCellTagHelper : TagHelper {

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            output.PreContent.SetHtmlContent("<b><i>");
            output.PostContent.SetHtmlContent("</i></b>
");
        }
    }
}
```

该标签助手处理拥有wrap属性的td元素，插入b和i元素以环绕输出元素的内容。代码清单23-23为Index.cshtml视图文件中的单元格添加了wrap属性。

代码清单23-23　在Views/Home文件夹下的Index.cshtml文件中添加HTML属性

```
@model IEnumerable<City>

@{ Layout = "_Layout"; }

<table class="table table-sm table-bordered">
    <thead class="bg-primary text-white">
        <tr>
            <th>Name</th>
            <th>Country</th>
            <th class="text-right">Population</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var city in Model) {
            <tr>
                <td wrap>@city.Name</td>
                <td>@city.Country</td>
                <td class="text-right">@city.Population
?.ToString("#,###")</td>
            </tr>
        }
    </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</
a>
```

如果运行应用程序，你会看到表格中第一列的单元格内显示的City对象被显示为加粗的斜体文

本。检查发送到浏览器的HTML内容，你将会看到如何通过PreContent与PostContent属性在源内容的前后添加内容，如下所示：

```
...
<tr>
    <td wrap><b><i>London</i></b></td>
    <td>UK</td>
    <td class="text-right">8,539,000</td>
</tr>
...
```

提　示

wrap属性保留在输出元素中。这是因为没有在标签助手类中定义这个属性的相关属性。如果希望阻止属性包含在输出中，可以在标签助手类中为它们定义属性，即使不需要属性的值。

23.3.3　使用依赖注入获取视图上下文数据

标签助手的一种常见用法是转换元素以便包含当前请求或当前视图模型的详细内容。作为示例，在Infrastructure/TagHelpers文件夹中添加名为FormTagHelper.cs的类文件，并用代码清单23-24所示的内容定义此类。

代码清单23-24　Infrastructure/TagHelpers文件夹下的FormTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    public class FormTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public FormTagHelper(IUrlHelperFactory factory)
 {
```

```
            urlHelperFactory = factory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContextData { get; set;
}
        public string Controller { get; set; }
        public string Action { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            IUrlHelper urlHelper = urlHelperFactory.Get
UrlHelper(ViewContextData);

            output.Attributes.SetAttribute("action", ur
lHelper.Action(
                Action ??
                    ViewContextData.RouteData.Values["a
ction"].ToString(),
            Controller ??
                ViewContextData.RouteData.Values["contr
oller"].ToString()));
        }
    }
}
```

顾名思义，FormTagHelper类处理form元素，设置action属性以指定将表单数据发送到何方。如

果form元素拥有controller和action属性，它们的属性值将用于生成目标URL；否则，将使用来自路由数据的controller和action值。

为了获取上下文数据，添加名为ViewContextData的属性，并使用两个特性进行修饰，如下所示：

```
...
[ViewContext]
[HtmlAttributeNotBound]
public ViewContext ViewContextData { get; set; }
...
```

ViewContext特性表示当创建FormTagHelper类的实例时，应该为相应的属性分配ViewContext对象，如第18章所示。ViewContext类提供了正在渲染的视图的详情、路由数据以及当前的HTTP请求，见第21章。

如果HTML元素input拥有view-context属性，那

么HtmlAttributeNotBound特性可防止MVC给该属性赋值。这种做法很好，特别是在开发其他开发者使用的标签助手时。

提　示

内置的用于表单的标签助手类可用于设定操作方法，并用于实际项目中。本书中的HTML helper仅用于演示上下文数据是如何使用的。请查看第24章以了解内置标签助手的详情。

标签助手可以在构造函数中定义作为服务的依赖项，它们将使用依赖注入特性进行解析。在本例

中，定义用于IUrlHelperFactory服务的依赖，以允许从路由数据创建传出的URL。在Process方法中，标签助手使用IUrlHelperFactory.GetUrlHelper方法来获取使用ViewContext对象配置的IUrlHelper对象，为输出元素的action属性创建URL。代码清单23-25展示了准备的视图，这里移除了action属性，以便使用标签助手来设置。

代码清单23-25　在Create.cshtml文件中删除表单元素的属性

```
@model City

@{ Layout = "_Layout"; }

<form method="post">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
```

```
    </div>
    <formbutton type="submit" bg-color="danger" />
    <formbutton type="reset" />
    <a bs-button-color="primary" href="/Home/Index">Can
cel</a>
</form>
```

如果运行应用程序并访问/Home/Create，然后检查发送到浏览器的HTML，你将会看到form元素拥有action属性，其值可使用上下文数据来设置，如下所示：

```
...
<form method="post" action="/Home/Create">
...
```

23.3.4　使用视图模型

标签助手还可以处理视图模型，裁剪执行的转换与创建的输出。为了进行演示，在Infrastructure/TagHelpers文件夹中创建名为LabelAndInputTagHelper.cs的类文件，并使用代码清单23-26所示内容定义此类。

代码清单23-26　Infrastructure/TagHelpers文件夹下的LabelAndInputTagHelper.cs文件的内容

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("label", Attributes = "helper-fo
r")]
    [HtmlTargetElement("input", Attributes = "helper-fo
r")]
    public class LabelAndInputTagHelper : TagHelper {

        public ModelExpression HelperFor { get; set; }
        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            if (output.TagName == "label") {
                output.TagMode = TagMode.StartTagAndEnd
Tag;
                output.Content.Append(HelperFor.Name);
                output.Attributes.SetAttribute("for", H
elperFor.Name);

            } else if (output.TagName == "input") {
                output.TagMode = TagMode.SelfClosing;
                output.Attributes.SetAttribute("name",
HelperFor.Name);
                output.Attributes.SetAttribute("class",
 "form-control");
                if (HelperFor.Metadata.ModelType == typ
```

```
eof(int?)) {
                    output.Attributes.SetAttribute("typ
e", "number");
                }
            }
        }
    }
}
```

该标签助手转换拥有helper-for属性的label与input元素。该标签助手的重要之处在于HelperFor属性的类型，该属性用于接收helperfor属性的值。

```
...
public ModelExpression HelperFor { get; set; }
...
```

当希望对视图模型的一部分进行处理时，将使用ModelExpression类。最简单的方式是跳过说明，如代码清单23-27所示，展示标签助手是如何应用于视图的。

代码清单23-27　在Views/Home文件夹下的Create.cshtml文件中应用标签助手处理模型

```
@model Cities.Models.City

@{  Layout = "_Layout"; }

<form method="post">
    <div class="form-group">
        <label helper-for="Name" />
        <input helper-for="Name" />
    </div>
    <div class="form-group">
        <label helper-for="Country" />
        <input helper-for="Country" />
    </div>
    <div class="form-group">
        <label helper-for="Population"/>
        <input helper-for="Population" />
    </div>
    <formbutton type="submit" bg-color="danger" />
    <formbutton type="reset" />
    <a bs-button-color="primary" href="/Home/Index">Can
cel</a>
</form>
```

helper-for属性的值来自Model类，该类由MVC检测并作为ModelExpression对象提供给标签助手。

本节不会深入说明ModelExpression类，因为对类型执行的任何检查都会导致无尽的类和属性列表。此外，MVC附带了一组有用的内置标签助手，

可使用视图模型来转换元素，如第24章所述，这意味着不必自己创建。

对于标签助手，这里使用了两个值得说明的基本特性。第一个特性是获取模型属性的名称，以便可以包含在输出元素中，如下所示：

```
...
output.Content.Append(HelperFor.Name);
output.Attributes.SetAttribute("for", HelperFor.Name);
...
```

Name属性返回模型属性的名称。

第二个特性是获取模型属性的类型，以便可以改变input元素的type属性，如下所示：

```
...
if (HelperFor.Metadata.ModelType == typeof(int?)) {
    output.Attributes.SetAttribute("type", "number");
}
...
```

如果执行应用程序并访问/Home/Create，然后

检查发送到浏览器的HTML内容，你将会看到如下元素：

```
<div class="form-group">
    <label for="Name">Name</label>
    <input name="Name" class="form-control" />
</div>
<div class="form-group">
    <label for="Country">Country</label>
    <input name="Country" class="form-control" />
</div>
<div class="form-group">
    <label for="Population">Population</label>
    <input name="Population" class="form-control" type=
"number" />
</div>
```

以上代码把用于input元素Population的type属性设置为number，以反映City.Population属性在C#类中为int类型。本节展示了标签助手如何反映不同的模型特征来生成HTML。基于使用的浏览器，input元素将只允许输入数字。

23.3.5　协调标签助手

TagHelperContext.Items属性提供了一个字典，用于在操作元素的标签助手与后继操作元素的标签助手之间进行协调。为了演示Items集合的用法，在Infrastructure/TagHelpers文件夹中添加名为CoordinatingTagHelpers.cs的类文件，在其中使用代码清单23-28所示的内容定义一对标签助手。

代码清单23-28　Infrastructure/TagHelpers文件夹下的CoordinatingTagHelpers.cs文件的内容

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("div", Attributes = "theme")]
    public class ButtonGroupThemeTagHelper : TagHelper
{

        public string Theme { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {
            context.Items["theme"] = Theme;
        }
    }
```

```
    [HtmlTargetElement("button", ParentTag = "div")]
    [HtmlTargetElement("a", ParentTag = "div")]
    public class ButtonThemeTagHelper : TagHelper {

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            if (context.Items.ContainsKey("theme")) {
                output.Attributes.SetAttribute("class",
                    $"btn btn-{context.Items["theme"]}"
);
            }
        }
    }
}
```

第一个标签助手是ButtonGroupThemeTagHelper类，用于处理拥有theme属性的div元素。协调标签助手可以转换自己的元素，但是这个示例只是简单地添加theme属性的值到Items字典，以便包含在div元素内的标签助手可以使用。

第二个标签助手是ButtonThemeTagHelper类，

用于处理拥有a元素和button元素的div元素，并使用来自Items字典的theme属性值来设置输出元素的Bootstrap样式。代码清单23-29展示了这些标签助手应用的元素集合。

代码清单23-29　在Views/Home文件夹下的Create.cshtml文件中应用协调标签助手

```
@model Cities.Models.City

@{  Layout = "_Layout"; }

<form method="post">
    <div class="form-group">
        <label helper-for="Name" />
        <input helper-for="Name" />
    </div>
    <div class="form-group">
        <label helper-for="Country" />
        <input helper-for="Country" />
    </div>
    <div class="form-group">
        <label helper-for="Population" />
        <input helper-for="Population" />
    </div>
    <div theme="primary">
        <button type="submit">Add</button>
        <button type="reset">Reset</button>
        <a href="/Home/Index">Cancel</a>
```

```
    </div>
</form>
```

如果运行应用程序并访问/Home/Create，你将会看到按钮组都以同样的方式修饰。如果修改div元素的theme属性为其他的Bootstrap主题设置，例如info、danger或primary，并重新加载页面，你将会看到修改已反映到按钮上，如图23-6所示。

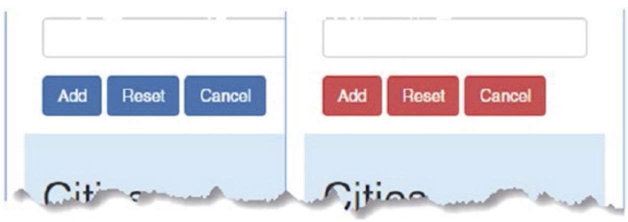


图23-6　协调标签助手

23.3.6　抑制输出元素

通过在使用Process方法的参数接收的TagHelperOutput对象上调用SuppressOuput方法，标签助手可以用于防止元素被包含在输出到浏览器的HTML中。代码清单23-30在共享布局中添加了一个

元素来显示高亮信息，但是这里仅仅对于特定的请求方式才显示。

代码清单23-30　在Views/Shared文件夹下的_Layout.cshtml文件中添加可视信息

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div show-for-action="Index" class="m-1 p-1 bg-dang
er">
        <h2>Important Message</h2>
    </div>
    <div title="Cities">@RenderBody()</div>
</body>
</html>
```

show-for-action属性指定了希望显示警告信息的操作方法的名称。这不是在实际应用程序中控制包含内容的有用方式，但是对于仅有一个控制器和

两个操作方法名称的示例应用来说是有效的。代码清单23-31展示了SelectiveTagHelper.cs类的内容，将它添加到Infrastructure/TagHelpers文件夹中。

代码清单23-31　Infrastructure/TagHelpers文件夹下的SelectiveTagHelper.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement(Attributes = "show-for-action")]
    public class SelectiveTagHelper : TagHelper {

        public string ShowForAction { get; set; }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }
        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {

            if (!ViewContext.RouteData.Values["action"]
.ToString()
                    .Equals(ShowForAction, StringCompar
```

```
ison.OrdinalIgnoreCase)) {
                    output.SuppressOutput();
                }
            }
        }
}
```

这个标签助手使用ViewContext从路由数据中获取action值，然后与HTML元素的show-for-action属性做比较。如果它们不匹配，则SuppressOutput方法被调用。为查看效果，启动应用程序并访问/Home/Index和/Home/Create。如图23-7所示，信息仅仅在Index这个操作方法被调用时显示。

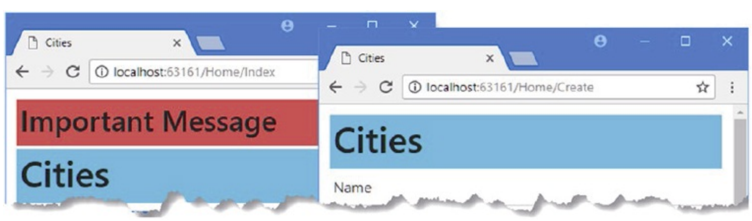


图23-7　使用标签助手抑制元素

23.4　小结

本章介绍了标签助手的用法，标签助手是

ASP.NET Core MVC的新增特性。本章还介绍了标签助手在Razor视图中的角色，并演示了自定义标签助手是如何创建、注册并应用的。本章接下来展示了如何控制标签助手的作用域，并说明了使用标签助手转换HTML元素的各种不同方式。下一章将说明标签助手在HTML表单元素中的使用。

第24章　使用表单标签助手

MVC提供了一套内置的标签助手来执行日常必需的HTML转换。本章介绍操作HTML表单的标签助手，HTML表单往往包括form、input、label、select、option和textarea元素。第25章将介绍其他内置的标签助手，它们提供非表单元素的特性。表24-1介绍了表单标签助手的背景。

表**24-1**　表单标签助手的背景

问题	答案
表单标签助手是什么？	表单标签助手用于转换HTML表单元素，从而不必编写自定义的标签助手就可以处理常见的多数问题
表单标签助手有何作用？	表单标签助手确保HTML表单元素（包括在表单内的元素，比如label和input元素）被一致地生成。大部分情况下，标签助手确保重要的属性（如id、name和for）已使用视图模型类正确设置，但是有些标签助手也可以生成内容，例如填充select元素中的option子元素

如何使用表单标签助手？	使用内置的标签助手查询带有asp-前缀的属性，比如asp-for属性
表单标签助手有何缺陷或限制？	仅有的限制是提供给标签助手的用以生成select元素中option子元素的模型数据的方式。24.5节将介绍这个问题，并提供自定义的标签助手来处理
表单标签助手有何替代品？	可以不使用这些标签助手在视图中编写HTML表单，也可以使用第23章介绍的技术，开发自己的标签助手

表24-2列出了本章要介绍的操作。

表**24-2**　本章要介绍的操作

操作	方法	代码清单
为form元素设置action属性	使用form元素标签助手	代码清单24-5
防止跨站请求伪造（cross-site request	为操作方法应用ValidateAntiForgeryToken特性，	代码清单24-6和

forgery）	在表单元素上设置asp-antiforgery属性（可选）	代码清单24-7
设置input元素的id、name和value属性	应用asp-for属性	代码清单24-8
格式化input元素显示的值	为input元素应用asp-format属性，或者在模型类上应用DisplayFormat特性	代码清单24-9～代码清单24-12
设置label元素的for属性和元素内容	应用asp-for属性	代码清单24-13
修改应用asp-for属性的label元素的内容	在模型类上应用Display特性，且使用Name属性指定内容	代码清单24-14
为select元素设置id和name属性	应用asp-for属性	代码清单24-15
生成option元素	应用asp-items属性	代码清单24-16～代码清单24-21
为textarea元素设置id和name属性	应用asp-for属性	代码清单24-22和代码清单24-23

24.1　准备示例项目

本章继续使用第23章创建的Cities项目。对于本章，需要启用来自MVC的内置标签助手，并禁用第23章创建的自定义标签助手。代码清单24-1展示了在视图导入文件中所做的变更，这里使用MVC的标签助手替换了@addTagHelper在Cities程序集中的标签助手，定义在命名空间Microsoft.AspNetCore.Mvc.TagHelpers中。

代码清单24-1　修改View文件夹下的_ViewImports.cshtml文件中的标签助手

```
@using Cities.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

重置视图和布局

代码清单24-2展示了Index.cshtml文件的内容，这里移除了使用自定义标签助手类的属性。

代码清单24-2　Views/Home文件夹下的Index.cshtml文件中的

```
@model IEnumerable<City>

@{  Layout = "_Layout"; }

<table class="table table-sm table-bordered">
    <thead class="bg-primary text-white">
        <tr>
            <th>Name</th>
            <th>Country</th>
            <th class="text-right">Population</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var city in Model) {
            <tr>
                <td>@city.Name</td>
                <td>@city.Country</td>
                <td class="text-right">@city.Population
?.ToString("#,###")</td>
            </tr>
        }
    </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</
a>
```

代码清单24-3展示了对Create.cshtml文件所做的相关修改，这里回到了标准的HTML元素，而没有使用第23章介绍的属性。

代码清单24-3　Views/Home文件夹下的Create.cshtml文件的内容

```
@model City

@{  Layout = "_Layout"; }

<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

最后修改共享布局，如代码清单24-4所示。

代码清单24-4　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，你将会看到城市列表，可以单击Create按钮，填充表单后提交新的数据给服务器，如图24-1所示。

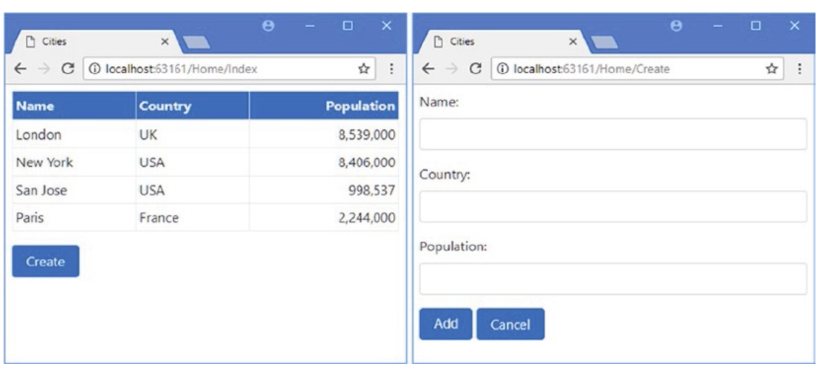


图24-1　运行示例应用

24.2　使用form元素

FormTagHelper类是form元素的内置标签助手，用于管理HTML表单的配置以便基于应用程序的路由配置指向正确的操作方法。该标签助手支持表24-3所示的属性。

表**24-3　form**元素的内置标签助手类定义的属性

属性	描述
asp-controller	用于指定action属性URL对应的路由系统的controller值。如果忽略，将使用呈现视图的控制器
asp-action	用于指定action属性URL对应的路由系统的action值。如果忽略，呈现视图的action将会被使用
asp-route-*	以asp-route-开头的属性名用于指定action属性URL的附加值，所以asp-route-id属性用来为路由系统提供id片段的值
asp-route	用于指定生成action属性URL的路由名称
asp-area	指定用于生成action属性URL的area名称
asp-antiforgery	控制是否为视图添加anti-forgery信息

24.2.1　设置form目标

FormTagHelper类的主要目的是使用应用程序的路由配置来设置form元素的action属性，确保即使路由架构发生了变化，表单数据也总是被发送到正确的URL地址。代码清单24-5使用asp-action和asp-controller属性来指向Home控制器的Create操作方法。

注　意

标签助手并不设置method属性，如果在form元素中忽略该属性，那么浏览器将会使用GET请求来发送表单数据。如第17章所述，如果表单数据用于修改应用程序中的数据，这可能导致问题。最佳实

践是设置method属性，尤其当希望使用GET请求时，显然不能忘了设置method属性为GET。

代码清单24-5　设置Views/Home文件夹下的Create.cshtml文件中的form目标

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" /
>
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
```

```
l</a>
</form>
```

如果运行应用程序，访问/Home/Create并检查发送到客户端的HTML，你将会看到标签助手已添加action属性到form元素中并使用路由系统设置了值，如下所示：

```
<form method="post" action="/Home/Create">
```

24.2.2 使用防伪特性

跨站请求伪造（Cross-Site Request Forgery，CSRF）是一种利用用户请求验证来攻击Web应用程序的方式。大多数Web应用程序（包括使用ASP.NET Core创建的）使用Cookie来标识哪些请求与特定会话相关，用户标识通常与之相关。

CSRF（也称为会话劫持）的详细说明参见维基百科，基于用户在使用应用程序，但是没有通过

单击Logout按钮显式结束会话之后访问恶意站点，应用程序仍然将用户会话视为活动的，存储在浏览器中的Cookie仍没有过期。恶意站点包含JavaScript代码，它向应用程序发送表单请求，在未经用户同意的情况下执行操作，操作的性质取决于被攻击的应用程序。由于JavaScript代码在用户浏览器中执行，因此向应用程序发出的请求包含会话的Cookie值，而应用程序会在用户不知情或不同意的情况下执行操作。

如果form元素没有包含action属性（因为是路由系统使用asp-controller和asp-acton属性生成的），那么FormTagHelper类将自动启用反CSRF特性，将安全令牌包含在表单的一个hidden类型的input元素中，并同Cookie一起发送到客户端。应用程序将仅仅处理同时包含Cookie和来自表单隐藏域的值的请求，恶意站点不能访问。表单生成的每个

请求都有新的唯一安全令牌。

如果运行应用程序，访问/Home/Create并查看发送到浏览器的HTML，你将会看到input隐藏域，如下所示：

```
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8KuVkH8hFlRApe
    FBxTrhCFTKZe0B9BKwnWDJqLRUDk__PrEwaeCJmiBbGkwW1ZI81
6c_TrM5XQkJBeqNI5IL8FhuO
    RvjZuYIL-GZvnWZ62OThsZYT02HNX_Lu5LWDNWDdVoS5O5hZtza
oHLeY5lNto" />
```

如果按F12键，就可以看到相关的添加到回应中的Cookie。将安全令牌添加到HTML响应中仅仅是处理的一部分，还必须在控制器中进行验证，如代码清单24-6所示。

代码清单24-6　在Colltrollers文件夹下的HomeController.cs中验证anti-forgery令牌

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;

```

```
namespace Cities.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Ci
ties);

        public ViewResult Create() => View();

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(City city) {
            repository.AddCity(city);
            return RedirectToAction("Index");
        }
    }
}
```

ValidateAntoForgeryToken特性确保请求中包含有效的反CSRF令牌，如果不存在或者没有包含期望的值，将会抛出异常。

FormTagHelper类提供了asp-antiforgery属性来覆写默认的反CSRF行为。如果该属性的值为true，

安全令牌将被包含到响应中，即使form元素拥有action属性。如果该属性的值为false，安全令牌将被禁用。代码清单24-7显式启用了该属性，因为没有在form元素中定义action属性，所以无论如何都将添加安全令牌。

代码清单24-7　在Views/Home文件夹下的Create.cshtml文件中启用反CSRF特性

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" name="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" name="Country" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" name="Population" />
```

```
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

提　示

测试反CSRF特性时需要一点技巧。首先让请求包含表单的URL地址（对于本例来说是/Home/Create），然后使用F12键来定位并删除表单中的input隐藏域。当填充表单后发送回应用程序时，浏览器不会提供请求的数据，请求将会失败并展示错误页面。

24.3　使用input元素

input元素是HTML表单的核心，它们提供了用户可以将非结构化数据提供给应用程序的主要方法。InputTagHelper类用于转换input元素，它们反映了在视图模型上获取的数据类型和格式，并使用表24-4所示的属性。

表**24-4**　用于**input**元素的内置标签助手的属性

属性	描述
asp-for	用于指定input元素表示的视图模型属性
asp-format	用于指定input元素表示的视图模型属性值的格式

24.3.1　配置input元素

asp-for属性被设置为视图模型属性的名称，从而用于设置input元素的id、type和value属性。代码

清单24-8为Create.cshtml文件中的input元素应用了asp-for属性。

代码清单24-8　在Views/Home文件夹下的Create.cshtml文件中配置input元素

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
                    asp-antiforgery="true">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" asp-for="Country" />

    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" asp-for="Population
" />
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
```

```
</form>
```

如果运行应用程序并访问/Home/Create，你将会看到标签助手使用asp-for属性来定制每个input元素，如以下代码所示（忽略了反CSRF安全令牌）：

```
<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" type="text" id="Name" name="Name" value="" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" type="text" id="Country"
               name="Country" value="" />
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" type="number" id="Population"
               name="Population" value="" />
    </div>
    <button type="submit" class="btn btn-primary">Add</button>
    <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

input元素的type属性指定了如何在表单中显示该元素。可以在用于Population属性的input元素中简单地看到结果，其type属性已经设置为number。因为C#的Population属性类型为int?，所以标签助手使用type属性来指示浏览器只有数字才能接受。

注　意

人们将type属性的解释方式留给了浏览器。并非所有的浏览器都响应HTML5规范中定义的所有类型，即使处理，它们的实现方式也存在差异。type属性是对表单中所需数据类型的有效提示，但是你应该使用模型验证特性来确保用户提供有用的数据。

表24-5说明了不同的C#属性类型和它们生成的input元素类型。

表**24-5　C#**属性类型和它们生成的**input**元素类型

C#属性类型	生成的**input**元素类型
byte、sbyte、int、uint、short、ushort、long、ulong	number
float、double、decimal	text，带有额外的用于模型验证的属性，稍后说明
bool	checkbox
string	text
DateTime	datetime

类型float、double和decimal能生成type属性为text的input元素，因为不是所有的浏览器都支持可

以用于这些类型的全部字符。为辅助用户，标签助手为input元素额外添加了用于模型验证的属性，具体将在第27章说明。

通过在input元素上定义type属性可以重写表24-5所示的默认映射。标签助手不会重写你定义的值，这允许你利用各种可用的input元素类型，例如password或hidden，以及HTML5中的新类型，比如number。

这种方式的不足是，必须在为给定的模型属性生成input元素的所有视图中设置type属性。如果需要在多个视图中覆写默认映射，可以为C#模型类的属性应用UIHint特性，指定表24-6中的值作为特性值。

____

提　示

如果模型属性不使用表24-5中的类型且没有使用UIHint特性进行装饰，标签助手将设置input元素的type属性为text。

表**24-6　UIHint**特性值及其生成的**input**元素类型

特性值	**input**元素类型
HiddenInput	hidden
Password	password
Text	text
PhoneNumber	tel
Url	url

EmailAddress	email
Time	time（用于显示DateTime对象的time部分）
Date	date（用于显示DateTime对象的date部分）
DateTime-local	datetime-local（用于显示没有提供时区信息的DateTime对象）

24.3.2　格式化数据

当操作方法提供带有视图模型的视图时，标签助手使用asp-for属性提供的值来设置input元素的value属性。asp-format属性用于指定数据如何格式化。

为了演示，在Home控制器中添加一个新的操作方法，如代码清单24-9所示。该操作方法从存储中选取第一个City对象，并作为Create视图的视图模型。

代码清单24-9　在Controller文件夹下的HomeController.cs文件

中添加操作方法

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;
using System.Linq;

namespace Cities.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Ci
ties);

        public ViewResult Edit() => View("Create", repo
sitory.Cities.First());

        public ViewResult Create() => View();

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(City city) {
            repository.AddCity(city);
            return RedirectToAction("Index");
        }
    }
}
```

如果运行应用程序，访问/Home/Edit并检查发

送给浏览器的HTML，你将会看到value属性已经使用视图模型对象填充了，如下所示：

```
<input class="form-control" type="number" id="Populatio
n"
       name="Population" value="8539000" />
```

asp-format属性接收一个将被传递给标准C#字符串格式化系统的值，如代码清单24-10所示。

代码清单24-10　在Views/Home文件夹下的Create.cshtml文件中格式化数据

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
                    asp-antiforgery="true">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" asp-for="Country" /
>
```

```
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" asp-for="Population
" asp-format="{0:#,###}" />
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

asp-format属性的值是逐字使用的，这意味着必须包含大括号与“0:引用”以及所需的格式。如果运行应用程序并访问/Home/Edit，你将看到Population值被格式化为如下形式：

```
<input class="form-control" type="number" id="Populatio
n"
       name="Population" value="8,539,000" />
```

asp-format属性需要慎用，因为必须确保应用程序的其他部分支持你所使用的格式。在这种情况下，你通过格式化Population的值引发了一个问题。标签助手设置input元素的type属性为number，

并对Population使用了表24-5中的默认映射，但是由指定的格式化串生成的value属性值包含非数字字符。因此，遵守number元素类型的浏览器（不是所有，记住）可能不会在元素中显示任何值。

你还必须确保应用程序能够解析你所使用的格式中的值。示例应用程序期望接收可以解析为 int类型的Population值，包含非数字字符将导致验证错误，如第27章所述。

通过模型类来指定格式

如果你总是希望对模型属性使用同样的格式，那么可以使用DisplayFormat特性修饰C#类，该特性定义在命名空间System.ComponentModel.DataAnnotations中。DisplayFormat特性需要两个参数来格式化数据的值：DataFormatString参数指定格式化串，

ApplyFormatInEditMode参数指定当值被编辑时格式化也应被使用。代码清单24-11使用DisplayFormat特性修饰了Population属性，使用的是可以被应用程序和浏览器处理的数字格式。

代码清单24-11　在Models文件夹下的City.cs文件中为模型类应用格式化特性

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {

    public class City {

        public string Name { get; set; }
        public string Country { get; set; }

        [DisplayFormat(DataFormatString = "{0:F2}", App
lyFormatInEditMode = true)]
        public int? Population { get; set; }
    }
}
```

属性asp-format优先于DisplayFormat特性，所以需要从视图中删除这个特性，如代码清单24-12

所示。

代码清单24-12　从Views/Home文件夹下的Create.cshtml文件中删除格式化特性

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
      asp-antiforgery="true">
    <div class="form-group">
        <label for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label for="Country">Country:</label>
        <input class="form-control" asp-for="Country" /
>
    </div>
    <div class="form-group">
        <label for="Population">Population:</label>
        <input class="form-control" asp-for="Population
" />
        </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

如果运行应用程序并访问/Home/Edit，你将看到Population值已经使用两位小数格式化，如下所示：

```
<input class="form-control" type="number" id="Populatio
n"
    name="Population" value="8539000.00" />
```

24.4　使用label元素

label元素可使用LabelTagHelper类进行转换，这个标签助手使用视图模型类来确保标签的免输入和一致性。支持的属性只有asp-for，该属性用于指定label元素表示的视图模型属性。

Label标签助手将使用视图模型属性的名称来设置for属性的值与label元素的内容。代码清单24-13在表单中为label元素应用了asp-for属性，它们将由标签助手进行转换。

代码清单24-13　在Views/Home文件夹下的Create.cshtml文件中应用Label标签助手

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
      asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <input class="form-control" asp-for="Country" />
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population" />
    </div>
    <button type="submit" class="btn btn-primary">Add</button>
    <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

因为label元素是空的，所以Label标签助手将使

用模型属性名称作为元素的内容并设置for属性，以告诉浏览器每个标签关联的input元素。如果运行示例应用程序，访问/Home/Create或/Home/Edit，并且检查发送到浏览器的HTML内容，你将看到如下输出元素：

```
<form method="post" action="/Home/Create">
    <div class="form-group">
        <label for="Name">Name</label>
        <input class="form-control" type="text" id="Nam
e"
               name="Name" value="London" />
    </div>
    <div class="form-group">
        <label for="Country">Country</label>
        <input class="form-control" type="text" id="Cou
ntry"
               name="Country" value="UK" />
    </div>
    <div class="form-group">
        <label for="Population">Population</label>
        <input class="form-control" type="number" id="P
opulation"
               name="Population" value="8539000.00" />
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

可以通过为模型类的属性应用Display特性来覆写作为label元素的内容，如代码清单24-14所示。

代码清单24-14　在Models文件夹下的City.cs文件中为模型属性改变说明信息

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {

    public class City {

        [Display(Name = "City")]
        public string Name { get; set; }

        public string Country { get; set; }

        [DisplayFormat(DataFormatString = "{0:F2}", App
lyFormatInEditMode = true)]
        public int? Population { get; set; }
    }
}
```

参数Name指定用于替代属性名称的值。如果运行示例，访问/Home/Create并检查发送给浏览器的HTML内容，你将会看到label元素的内容已经改

变，如下所示：

```
<div class="form-group">
    <label for="Name">City</label>
    <input class="form-control" type="text" id="Name" n
ame="Name" value="London" />
</div>
```

注意，for属性的值没有变化，所以浏览器知道label元素已关联到特定的input元素，且不受Display特性影响。

提　示

可以通过自定义label元素的内容来阻止标签助手进行设置。如果希望label元素不只包含属性的名称，这将很有用，这就是Label标签助手可以提供的

全部内容。

24.5　使用select和option元素

select和option元素为用户提供固定的选择集合，而不像input元素那样支持所有可能的开放数据选项。SelectTagHelper类负责转换select元素，支持的属性如表24-7所示。

表**24-7**　用于**select**元素的内置标签助手类定义的属性

属性	描述
asp-for	用于指定select元素表示的视图模型属性
asp-items	用于指定包含在select元素中的option元素值的来源

属性asp-for用于设置for和id属性以反映接收的模型属性。代码清单24-15使用定义了asp-for属性的

select元素来替换用于Country属性的input元素。

代码清单24-15　在Views/Home文件夹下的Create.cshtml文件中使用select元素

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
      asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <select class="form-control" asp-for="Country">
            <option disabled selected value="">Select a
 Country</option>
            <option>UK</option>
            <option>USA</option>
            <option>France</option>
            <option>China</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population
" />
    </div>
```

```
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

这里手动填充了select元素的option元素，以提供用户选择的国家范围。如果运行应用程序并访问/Home/ Create，你将看到发送给浏览器的HTML包含如下select元素：

```
<select class="form-control" id="Country" name="Country
">
    <option disabled selected value="">Select a Country
</option>
    <option>UK</option>
    <option>USA</option>
    <option>France</option>
    <option>China</option>
</select>
```

如果访问/Home/Edit并检查发送到浏览器的HTML内容，你将看到视图模型属性Country的值已经被修改为选中的option元素，如下所示：

```
<select class="form-control" id="Country" name="Country
```

```
">
    <option disabled selected value="">Select a Country
</option>
    <option selected="selected">UK</option>
    <option>USA</option>
    <option>France</option>
    <option>China</option>
</select>
```

选中option元素的任务是由OptionTagHelper类执行的，这个标签助手通过TagHelperContext.Items集合接收来自SelectTagHelper的指令。如第23章所述，标签助手使用的集合需要协同工作，当创建自定义的标签助手以解决内置的限制时，可利用通过SelectTagHelper添加到Items集合中的数据。

24.5.1　使用数据源填充select元素

显式定义select元素的option元素，对于选择始终具有同样可能性的情况是有帮助的。但是，当需要提供从数据模型中获取选项，或当需要在多个视图中具有相同的选项集合并且不希望手动维护重复

的内容时，就没有什么用了。

24.5.2　从枚举中生成option元素

如果有固定的选项集合呈现给用户，并且不希望在整个应用程序的视图中重复它们，那么可以使用枚举。在Models文件夹中添加名为CountryNames.cs的类文件，并用它定义代码清单24-16所示的枚举。

代码清单24-16　Models文件夹下的CountryNames.cs文件的内容

```
namespace Cities.Models {

    public enum CountryNames {
        UK,
        USA,
        France,
        China
    }
}
```

不能直接在asp-item属性中使用枚举，因为标

签助手期望处理的是SelectListItem对象序列。但是，可以使用辅助方法执行必要的转换，如代码清单24-17所示。

代码清单24-17　在Views/Home文件夹下的Create.cshtml文件中使用枚举生成option元素

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
      asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <select class="form-control" asp-for="Country"
                asp-items="@new SelectList(Enum.GetName
s(typeof(CountryNames)))">
            <option disabled selected value="">Select a
 Country</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population
```

```
" />
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

当使用枚举时，生成option元素的最佳方式，是为asp-items属性提供由枚举值的名称填充的SelectList对象。在幕后，SelectTagHelper类从IEnumerable<SelectListItem>生成option元素，SelectList类实现了这个接口。

如果运行应用程序并访问/Home/Create或/Home/Edit，你将看到发送到浏览器的HTML中包含与枚举值对应的option元素，如下所示：

```
<select class="form-control" id="Country" name="Country
">
    <option disabled selected value="">Select a Country
</option>
    <option>UK</option>
    <option>USA</option>
    <option>France</option>
    <option>China</option>
```

```
</select>
```

注意，标签助手保留了占位用的option元素。任何显式定义的option元素都保持不变，这意味着不必在数据中混合占位符。

1．通过模型生成**option**元素

如果需要生成option元素以反映模型中的数据，那么最简单的方式是通过View Bag来提供生成元素所需的数据，如代码清单24-18所示。

代码清单24-18　在Controllers文件夹下的HomeController.cs中通过View Bag提供数据

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Cities.Controllers {

    public class HomeController : Controller {
        private IRepository repository;
```

```
        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Ci
ties);

        public ViewResult Edit() {
            ViewBag.Countries = new SelectList(reposito
ry.Cities
                .Select(c => c.Country).Distinct());
            return View("Create", repository.Cities.Fir
st());
        }

        public ViewResult Create() {
            ViewBag.Countries = new SelectList(reposito
ry.Cities
                .Select(c => c.Country).Distinct());
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(City city) {
            repository.AddCity(city);
            return RedirectToAction("Index");
        }
    }
}
```

操作方法Edit和Create将ViewBag.Countries属性设置为使用存储库中的City.Country唯一值填充的

SelectList对象。代码清单24-19使用asp-items属性告诉标签助手为option元素从Countries属性中获取数据。

代码清单24-19　在Views/Home文件夹下的Create.cshtml文件中为option元素使用View Bag

```
@model City

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
      asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <select class="form-control" asp-for="Country"
asp-items="ViewBag.Countries">
            <option disabled selected value="">Select a
 Country</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population" />
```

```
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

如果运行应用程序并访问/Home/Create或/Home/Edit，你将看到生成的option元素，如下所示：

```
<select class="form-control" id="Country" name="Country
">
    <option disabled selected value="">Select a Country
</option>
    <option selected>UK</option>
    <option>USA</option>
    <option>France</option>
</select>
```

2．使用自定义的标签助手从模型生成**option**元素

通过View Bag为option元素传递数据的问题在于，必须在每个渲染视图的操作方法中留意为标签助手生成数据，这将导致代码的重复。你可以在代

码清单24-18中获取一些感觉，这使得测试和维护控制器变得更加困难。

更好的方式是创建自定义的标签助手来支持内置的SelectTagHelper类。在Infrastructure/TagHelper文件夹中创建名为SelectOptionTagHelper.cs的类文件，定义代码清单24-20所示的类。

代码清单24-20　Infrastructure/TagHelper文件夹下的SelectOptionTagHelper.cs文件的内容

```
using System;
using System.Linq;
using System.Reflection;
using System.Threading.Tasks;
using Cities.Models;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

    [HtmlTargetElement("select", Attributes = "model-fo
r")]
    public class SelectOptionTagHelper : TagHelper {
        private IRepository repository;

        public SelectOptionTagHelper(IRepository repo)
```

```
{
            repository = repo;
        }

        public ModelExpression ModelFor { get; set; }

        public override async Task ProcessAsync(TagHelp
erContext context,
                TagHelperOutput output) {

            output.Content.AppendHtml(
                (await output.GetChildContentAsync(fals
e)).GetContent());

            string selected = ModelFor.Model as string;

            PropertyInfo property = typeof(City)
                .GetTypeInfo().GetDeclaredProperty(Mode
lFor.Name);
            foreach (string country in repository.Citie
s
                    .Select(c => property.GetValue(c)).
Distinct()) {
                if (selected != null && selected.Equals
(country,
                        StringComparison.OrdinalIgnoreC
ase)) {
                    output.Content
                        .AppendHtml($"<option selected>
{country}</option>");
                } else {
                    output.Content.AppendHtml($"<option
>{country}</option>");
                }
            }
            output.Attributes.SetAttribute("Name", Mode
```

```
lFor.Name);
            output.Attributes.SetAttribute("Id", ModelF
or.Name);
        }
    }
}
```

该标签助手通过model-for属性在select元素上执行操作，并使用依赖注入接收可以不依赖渲染视图的来自控制器的用于访问模型数据的存储库对象。该标签助手定义了异步的ProcessAsync方法，从而简化了获取和保留select元素中任何现有内容的过程，这是通过GetChildContentAsync方法实现的。

SelectTagHelper指示应通过Items集合中的选项（使用自己的类型为键）选择option元素的名称。该标签助手获取已选中选项的列表，并与LINQ查询结果结合使用，以便为存储库中的每个唯一值生成option元素。代码清单24-21更新了select元素，以便asp-items属性被model-for属性替换，还添加了

@addTagHelper表达式，从而仅为Create视图启用自定义的标签助手。

代码清单24-21　在Views/Home文件夹下的Create.cshtml文件中启用自定义的标签助手

```
@model City
@addTagHelper Cities.Infrastructure.TagHelpers.SelectOp
tionTagHelper, Cities

@{  Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
    asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <select class="form-control" model-for="Country
">
            <option disabled selected value="">Select a
 Country</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population
" />
```

```
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

新的标签助手会生成同样的内容，但是不需要内置标签助手要求的View Bag数据。这种方式让操作方法可以专注于它们的特定任务，并保持应用程序的整体外观。

24.6 使用textarea元素

textarea元素用于从用户获取更多的文本，通常用于非结构化数据，比如注释或观测结果。TextAreaTagHelper负责转换textarea元素，并支持单个属性asp-for，该属性用于指定textarea元素表示的视图模型属性。

TextAreaTagHelper类相对简单，为asp-for属性

提供的值用于为textarea元素设置id和name属性。为了演示这个标签助手，为City模型类添加了一个新的属性，如代码清单24-22所示。

代码清单24-22　在Models文件夹下的City.cs文件中添加属性

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {

    public class City {

        [Display(Name = "City")]
        public string Name { get; set; }

        public string Country { get; set; }

        [DisplayFormat(DataFormatString = "{0:F2}", App
lyFormatInEditMode = true)]
        public int? Population { get; set; }

        public string Notes { get; set; }
    }
}
```

代码清单24-23在Create.cshtml视图文件中添加了一个textarea元素，并使用asp-for属性关联这个元

素到City类的Notes属性

代码清单24-23　在Views/Home文件夹下的Create.cshtml文件中添加文本区域

```
@model City
@addTagHelper Cities.Infrastructure.TagHelpers.SelectOp
tionTagHelper, Cities

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="C
reate"
      asp-antiforgery="true">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Country"></label>
        <select class="form-control" asp-for="Country"
asp-items="ViewBag.Countries">
            <option disabled selected value="">Select a
 Country</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Population"></label>
        <input class="form-control" asp-for="Population
" />
    </div>
    <div class="form-group">
```

```
        <label asp-for="Notes"></label>
        <textarea class="form-control" asp-for="Notes">
</textarea>
    </div>
    <button type="submit" class="btn btn-primary">Add</
button>
    <a class="btn btn-primary" href="/Home/Index">Cance
l</a>
</form>
```

如果运行应用程序并访问/Home/Create或/Home/Edit。你将看到发送给浏览器的HTML包含如下所示的textarea元素：

```
<div class="form-group">
    <label for="Notes">Notes</label>
    <textarea id="Notes" name="Notes"></textarea>
</div>
```

TextAreaTagHelper类相对简单，但它提供了本章介绍的其他表单标签助手的一致性。

24.7 验证表单标签助手

与HTML表单相关的其他标签助手有两个（见

表24-8），将在第27章详加说明。这两个标签助手用于在用户提供的数据不满足应用程序的期望时向用户提供反馈。

表**24-8**　验证表单标签助手

名称	描述
ValidationMessage	用于提供单个表单元素的验证反馈
ValidationSummary	用于提供关于表单所有元素的验证反馈

24.8　小结

本章介绍了用于转换HTML表单元素的内置标签助手。这些标签助手确保表单直接从模型类生成，这减少了潜在的错误并提供一致的方式来编写Razor视图。下一章将介绍其他内置标签助手，它们用于对一系列各不相同的HTML元素进行操作。

第25章　使用其他内置标签助手

第24章介绍的标签助手专注于生成HTML表单，但这不是仅有的ASP.NET Core MVC内置标签助手。本章将介绍用于管理JavaScript脚本与CSS样式表，为超链接元素创建URL，为图片元素提供缓存清除，以及支持数据缓存的标签助手。本章还会介绍提供应用程序相关的URL支持的标签助手，以确保在应用程序被发布到与其他应用程序共享的环境中时，浏览器可以访问静态内容。表25-1列出了本章要介绍的操作。

表25-1　本章要介绍的操作

操作	方法	代码清单
包含基于宿主环境的内容	使用environment元素	代码清单25-2和代码清单25-8

选择JavaScript文件	为script元素应用asp-src-include与asp-src-exclude属性	代码清单25-3～代码清单25-7
为JavaScript使用CDN	为script元素应用asp-fallback属性	代码清单25-9和代码清单25-10
选择CSS文件	为link元素应用asp-href-include与asp-href-exclude属性	代码清单25-11
为CSS文件使用CDN	为link元素应用asp-fallback属性	代码清单25-12
为anchor元素生成URL	使用AnchorTagHelper	代码清单25-13
确保图像的修改被检测	为img元素应用asp-append-version属性	代码清单25-14
缓存数据	使用cache元素	代码清单25-15～代码清单25-23
创建应用程序相关的URL	使用~字符添加URL前缀	代码清单25-24～代码清单25-26

25.1　准备示例项目

本章继续使用第24章的Cities项目，创建

wwwroot/images文件夹，并在其中添加名为city.png的图片文件。这是一张纽约天际线的公用全景照片，如图25-1所示。



图25-1　添加图片到项目中

这幅图片包含在本章的源代码中。如果需要下载示例应用，也可以替换为自己的图片。

本章要做的其他调整是使用Bower将jQuery加入项目中，如代码清单25-1所示。

代码清单25-1　在Cities文件夹下的bower.json文件中添加jQeury

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
```

```
    }
}
```

如果运行应用程序，你将能够列出存储库中的对象，并创建新的对象，如图25-2所示。

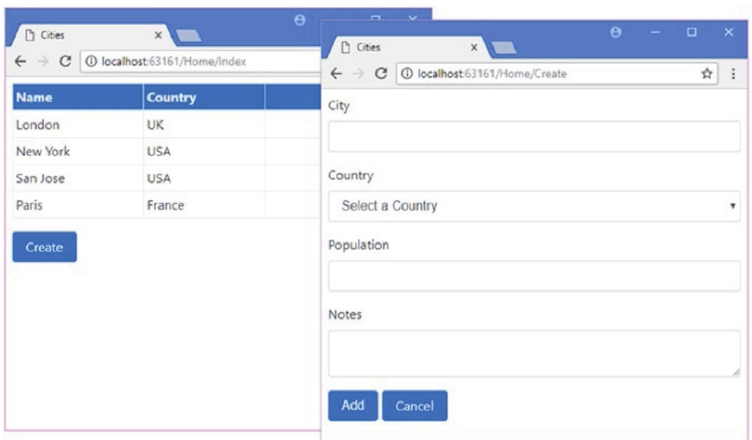


图25-2　运行示例应用

25.2　使用宿主环境标签助手

通过将EnvironmentTagHelper类应用于自定义的environment元素，可基于宿主环境决定是否将一块内容发送到浏览器，第14章对此曾经介绍过。开始时这可能看起来不太激动人心，但是随后需要借

助这个标签助手以充分利用一些相关的功能。environment元素依赖于names属性，该属性用于指定一个以逗号分隔的宿主环境名称列表，包含在environment元素中的内容将被包含在发送到客户端的HTML中。

代码清单25-2在共享的布局文件中添加了environment元素，对于开发和生产环境，分别在视图中包含不同的内容。

代码清单25-2　在Views/Shared文件夹下的_Layout.cshtml中使用environment元素

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <environment names="development">
        <div class="m-1 p-1 bg-info"><h2>This is Develo
```

```
pment</h2></div>
    </environment>
    <environment names="production">
        <div class="m-1 p-1 bg-danger"><h2>This is Prod
uction</h2></div>
    </environment>
    <div>@RenderBody()</div>
</body>
</html>
```

图25-3展示了在开发和生产宿主环境下运行应用程序的效果。environment元素检查当前的宿主环境名称，以及是否包含或忽略内容（environment元素本身不会发送到客户端的HTML中）。

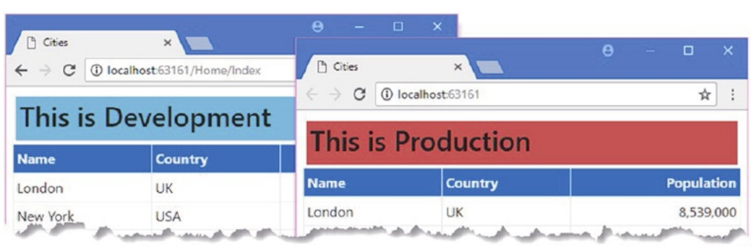


图25-3　使用宿主环境管理内容

25.3　使用JavaScript和CSS标签助手

另一类内置的标签助手用来通过script和link元

素管理JavaScript和CSS样式表。通常它们包含于共享布局中。这些标签助手强大而灵活，但是需要密切注意以避免产生意外的结果。

25.3.1　管理JavaScript文件

内置的ScriptTagHelper类是用于script元素并在视图中管理JavaScript文件的标签助手，可使用表25-2所示的属性包含JavaScript文件，随后将进行说明。

表**25-2**　用于**script**元素的内置标签助手类定义的属性

属性	描述
asp-src-include	用于指定JavaScript文件将被包含在视图中
asp-src-exclude	用于指定JavaScript文件将被排除在视图之外
asp-append-version	用于缓存清除
asp-fallback-src	用于指定在使用CDN有问题时，需要使用的回退JavaScript文件

asp-fallback-src-include	用于指定在使用CDN有问题时，将被使用的JavaScript文件
asp-fallback-src-exclude	用于指定当CDN有问题时，将被排除的JavaScript文件
asp-fallback-test	用于指定一段JavaScript，从而判断JavaScript代码是否已从CDN正确下载

1．选择JavaScript文件

属性asp-src-include使用通配符在视图中包含JavaScript文件。如第7章所述，通配符模板支持一系列通配符用于匹配文件。表25-3描述了常用的通配符。

表**25-3**　常用的通配符

通配符	示例	描述
?	js/src?.js	匹配除/之外的任意单个字符。示例匹配包含于js文件夹中，能匹配名称为src后跟任意一个字符且以.js结尾的任何文件，例如js/src1.js和js/srcX.js，但是不匹配js/src123.js或js/mydir/src1.js

*	js/*.js	匹配除/之外任意数量的任意字符。示例匹配包含于js文件夹中，能匹配以.js为扩展名的任意文件，例如js/src1.js和js/src123.js，但是不包括js/mydir/src1.js
**	js/**/*.js	匹配包含/的任意数量的任意字符。示例匹配包括js目录及其子目录中以.js为扩展名的任何文件，例如/js/src1.js和/js/mydir/src1.js

通过asp-src-include属性使用通配符模式，意味着应用程序中的视图将总是包含这些JavaScript文件，即使文件的名称或路径发生变化，以及文件被添加或被删除。代码清单25-3选择了jQuery包中的文件，可使用Bower将jQuery安装到wwwroot/lib/jquery/dist文件夹中。

代码清单25-3　在Views/Shared文件夹下的_Layout.cshtml文件中选择JavaScript文件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script asp-src-include="/lib/jquery/dist/**/*.js">
</script>
```

```
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

以上示例中使用的通配符很常用。通配符在wwwroot文件夹中被计算，其中jQuery库可作为单个名为jquery.js的文件发布。

以上通配符试图选择文件，以适应jQuery在未来发布中的变化，例如改变JavaScript文件名称。如果运行示例应用程序并检查发送到浏览器的HTML内容，你将会发现里面包含错误，如下所示：

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="/lib/jquery/dist/core.js"></script>
    <script src="/lib/jquery/dist/jquery.js"></script>
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <script src="/lib/jquery/dist/jquery.slim.js"></scr
ipt>
```

```
    <script src="/lib/jquery/dist/jquery.slim.min.js"><
/script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

ScriptTagHelper类为匹配通过asp-src-include属性传递的通配符的每个文件生成一个script元素，而不只是选择jquery.js文件，jquery.min.js是紧缩版本的jquery.js文件，此处还有正常和紧缩版本的jQuery库core和slim。

由于Visual Studio默认隐藏了，因此你可能没有意识到jQuery发行版包含紧缩文件。为了展现wwwroot/ lib/jquery/dist文件夹中的全部内容，必须在Solution Explorer窗格中展开jquery.js，然后同样展开jquery.min.js，如图25-4所示。

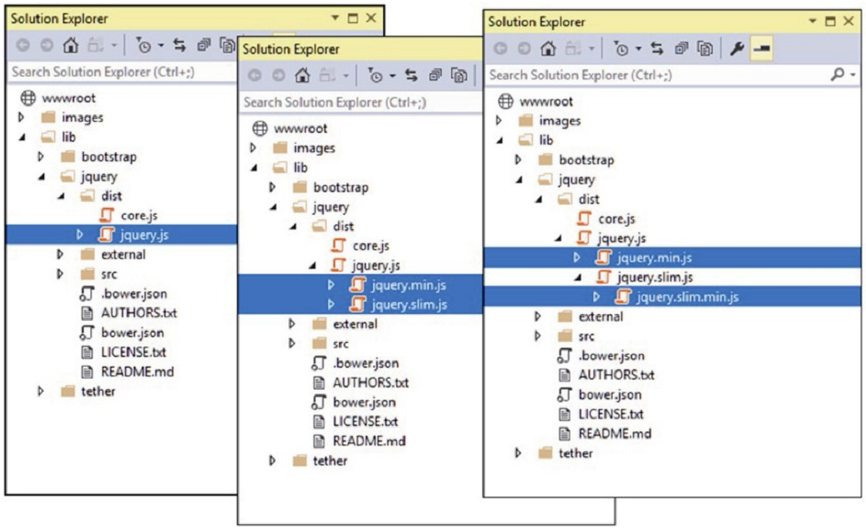


图25-4　在Solution Explorer窗格中展现目录的填充内容

代码清单25-3中使用的通配符会将jQuery代码加入浏览器多次，这不仅浪费带宽而且拖慢了应用程序的下载速度。对于某些库，可能导致错误或未预期的行为。解决这个问题有3种方式，这些方式随后进行说明。

使用源码映射

JavaScript文件通过紧缩来减少尺寸，这意味着

可以更快地发送到客户端，使用更少的带宽。紧缩过程会从源文件中删除所有的空白，重命名函数和变量，比如myHelpfullyNamedFunction可使用更少数量的字符来表示，例如x1。当在紧缩代码中使用浏览器的调试器来追踪错误时，x1这样的名字使得几乎不可能进行代码跟踪。

jquery.min.map 文件是源码映射文件，有些浏览器可以用它提供紧缩代码与开发者可读的未紧缩源码间的映射。

在写作本书时，源码映射还不是通用的支持特性，但可以在最新版本的Chrome和Edge浏览器中使用。例如，在Chrome浏览器中，如果打开开发者工具窗口，浏览器将自动请求源码映射文件，这意味着在发送紧缩版本的JavaScript文件时，仍然可以轻松启用调试。

2．限制通配符

许多库同时提供正常和紧缩版本的JavaScript文件，如果仅仅希望使用紧缩版本，那么可以限制通配符匹配的文件，如代码清单25-4所示。如果并不期望使用调试版本的jQuery库，因为它们编写良好，问题很少，或者知道浏览器支持源码映射，那么这是一种不错的方式。

代码清单25-4　在Views/Shared文件夹下的_Layout.cshtml文件中仅仅使用紧缩版本的JavaScript文件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script asp-src-include="/lib/jquery/dist/**/*.min.
js"></script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
```

```
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序并检查发送到浏览器的HTML，你将会看到仅包含紧缩版本的jQuery文件。

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="/lib/jquery/dist/jquery.min.js">
    </script><script src="/lib/jquery/dist/jquery.slim.
min.js"></script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

虽然限制通配符会起到作用，但是浏览器仍然最终发送了正常和轻量版本的jQuery库（轻量版本忽略了一些不常用的功能，详见jQuery网站）。为了进一步限制，可以在统配模板中只包含轻量版

本，如代码清单25-5所示。

代码清单25-5　在Views/Shared文件夹下的_Layout.cshtml文件中做进一步限制

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script asp-src-include="/lib/jquery/dist/**/*.slim
.min.js"></script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

结果是只有一个版本的jQuery文件被发送到浏览器中，但仍然保持了文件位置的灵活性。

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="/lib/jquery/dist/jquery.slim.min.js"><
```

```
/script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

1）排除文件

当希望选择的文件名中包含特定术语时，限制通配符是有帮助的，但是在没有特定术语的时候，就没有什么用了。比如，当希望包含完全版本的紧缩文件时。幸运的是，可以使用asp-src-exclude属性，从asp-src-include属性匹配的文件中移除文件，如代码清单25-6所示。

代码清单25-6　在Views/Shared文件夹下的_Layout.cshtml文件中排除文件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script asp-src-include="/lib/jquery/dist/**/*.min.
js"
```

```
        asp-src-exclude="**.slim.**">
    </script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

运行示例应用程序并检查发送到浏览器的HTML，你将会看到只包含完全紧缩版本的JavaScript文件。

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

同样的技术也可以用于包含非紧缩版本的文件，在开发时这很有用，如代码清单25-7所示。

代码清单25-7　在Views/Shared文件夹下的_Layout.cshtml文件

中选取非紧缩文件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script asp-src-include="/lib/jquery/dist/**/j*.js"
            asp-src-exclude="**.slim.**,**.min.**">
    </script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

注意，可以通过逗号分隔来指定多个术语。如果运行示例应用程序并检查发送到浏览器的HTML，你将会看到只有非紧缩版本的JavaScript文件被包含：

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="/lib/jquery/dist/jquery.js"></script>
```

```
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

2）使用宿主环境来选择文件

常见的工作方式是在开发过程中使用常规版本的JavaScript文件，这使调试变得简单，然后在生产环境中使用紧缩版本的JavaScript文件，从而节省了带宽。这可以通过使用environment元素，基于宿主环境选择包含的script元素来实现，如代码清单25-8所示。

代码清单25-8　在Views/Shared文件夹下的_Layout.cshtml中使用宿主环境来选择文件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <environment names="development">
        <script asp-src-include="/lib/jquery/dist/**/j*
.js"
```

```
                asp-src-exclude="**.slim.**,**.min.**">
        </script>
    </environment>
    <environment names="staging, production">
        <script asp-src-include="/lib/jquery/dist/**/*.
min.js"
                asp-src-exclude="**.slim.**">
        </script>
    </environment>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

这种方式的优势在于可基于宿主环境适配应用，但也意味着不得不编写和维护多个script元素。

缓存清除

可经常缓存静态内容（如图片、CSS样式表以及JavaScript文件）以停止从应用服务器请求很少变化的内容。缓存可以通过多种途径完成：浏览器可

以被告知缓存来自服务器的内容，应用程序可以使用缓存服务器来支持应用服务器，或者通过内容分发网络来分发内容。不是所有的缓存都在你的控制之下。例如，大的企业经常安装缓存来减少带宽，因为后继请求趋向于访问同样的站点或应用程序。

缓存的问题是，客户端在部署时不会立即收到新版本的静态文件，因为它们的请求仍由以前缓存的内容处理。最终，缓存的内容将过期，并且将使用新内容，但这会留下一段时间，在这段时间，应用程序控制器生成的动态内容与缓存交付的静态内容不同步。根据更新的内容，这可能导致布局问题或应用程序异常。

解决这个问题的方法称为缓存清除。思路是允许缓存处理静态内容，但是当内容在服务器上被修改之后立即做出反应。标签助手通过在用于静态内容的URL查询串后包含校验和作为版本来支持。例

如，ScriptTagHelper类通过asp-append-version属性来支持缓存清除。

```
...
<script asp-src-include="/lib/jquery/dist/**/j*.js"
        asp-src-exclude="**.slim.**,**.min.**"
        asp-append-version="true">
</script>
...
```

启用缓存清除导致生成的发送到浏览器的HTML元素如下所示：

```
...
<script src="/lib/jquery/dist/jquery.min.js?v=3zRSQ1HF-ocUiVcdv9yKTXqM">
</script>
...
```

同样的版本号也可以用于标签助手，直至改变文件的内容，比如更新JavaScript库，此刻另一个不同的校验和将被计算出来。附加版本号意味着每次修改文件时，客户端将请求不同的URL，此前缓存的内容无法达成，缓存视为对新内容的请求并发送

到应用服务器。内容将正常缓存直到下一次更新，这将生成另一个具有不同版本的URL。

3．使用**CDN**

内容分发网络（Content Delivery Network，CDN）用于将用户请求分流到离用户更近的服务器。浏览器不是从应用服务器请求JavaScript文件，而是从解析为本地服务器的主机来请求，这减少了加载文件所需的时间，降低了为应用程序提供的带宽。如果拥有庞大的、广域分布的用户群，那么可以使用商业注册的CDN，但是即使对于最小和最简单的应用程序，也可以通过受益于由主流技术公司管理的免费CDN来分发通用的JavaScript包，例如jQuery。

本章将使用微软提供的CDN，它们对流行的包提供免费访问，可以在ASP.NET网站上找到微软提供的CDN清单。对于jQuery 3.2.1，有6个包：

- jquery-3.2.1.js；
- jquery-3.2.1.min.js；
- jquery-3.2.1.min.map；
- jquery-3.2.1.slim.js；
- jquery-3.2.1.slim.min.js；
- jquery-3.2.1.slim.min.map。

这些包可以为完全版本和轻量版本提供常规的JavaScript文件、紧缩的JavaScript文件以及紧缩文件的源码映射文件。代码清单25-9通过修改示例应用中的布局文件来使用CDN获取的紧缩文件替换本地文件。

代码清单25-9　在Views/Shared文件夹下的_Layout.cshtml中使用CDN

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="****://ajax.aspnetcdn.***/ajax/jQuery/
jquery-3.2.1.min.js"></script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

指定CDN意味着对于jQuery文件没有请求到达应用服务器。使用CDN的问题是CDN不在你的控制之下，这意味着CDN可能失效，只剩下应用程序在运行，但是不能如预期那样工作，因为CDN内容不可用。为了帮助解决这一问题，ScriptTagHelper类提供了在CDN内容不能加载到客户端时回退到本地文件的能力，如代码清单25-10所示。

代码清单25-10　在Views/Shared文件夹下的Layout.cshtml中使用CDN回退

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="****://ajax.aspnetcdn.***/ajax/jQuery/
jquery-3.2.1.min.js"
            asp-fallback-src-include="/lib/jquery/dist/
**/*.min.js"
            asp-fallback-src-exclude="**.slim.**"
            asp-fallback-test="window.jQuery">
    </script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

属性asp-fallback-src-include与asp-fallback-src-exclude用于在CDN不能通过常规的src属性分发内容时选择和排除本地文件。为了判断CDN是否正常工作，asp-fallback-test属性用于定义在浏览器端执行的JavaScript片段，如果执行结果为false，那么回退文件将被请求。

运行应用程序并检查发送到客户端的HTML，你将看到ScriptTagHelper类从asp-fallback-test属性取出JavaScript片段并用它生成另一个script元素，如下所示：

```
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <script src="****://ajax.aspnetcdn.***/ajax/jQuery/
jquery-3.2.1.min.js">
    </script>
    <script>
        (window.jQuery||document.write("\u003Cscript
           src=\u0022\/lib\/jquery\/dist\/jquery.min.js
           \u0022\u003E\u003C\/script\u003E"));
    </script>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
```

如果文件从CDN加载，那么你在asp-fallback-test属性中指定的JavaScript片段必须返回true；否则，返回false。最简单的方式通常是检查JavaScript代码提供的功能入口点。jQuery库在公共的window

对象上创建了名为jQuery的函数，这也是代码清单25-10要测试的内容。你需要为从CDN加载的每个文件找到等价的测试。

测试回退设置也很重要，因为在CDN停止工作导致用户无法访问应用程序之前，无法发现CDN是否失效。最简单的检查回退的方式是将src属性指定的文件名改为某个不存在的文件名（可在文件名后追加FAIL），然后使用F12键来查看浏览器的网络请求。你应该看到CDN文件出错了，后面跟着对回退文件的请求。

警　告

CDN回退特性依赖于浏览器同步加载和执行

script元素，并按照定义的顺序依次执行。有很多异步处理技术可用来加速JavaScript文件的加载和执行，但是它们将导致在浏览器从CDN获取文件之前执行回退测试，结果导致在CDN完美工作时，首先淘汰CDN的使用。不要和CDN回退特性一起混用异步脚本加载。

25.3.2　管理CSS样式表

LinkTagHelper类是内置的作用于link元素的标签助手，用来管理视图中包含的CSS。该标签助手支持的属性如表25-4所示。

表**25-4**　用于**link**元素的内置标签助手类定义的属性

属性	描述
asp-href-include	用于为输出元素的href属性选择文件

asp-href-exclude	用于为输出元素的href属性排除文件
asp-append-version	用于启用缓存清除
asp-fallback-href	用于在CDN出现问题时指定回退文件
asp-fallback-href-include	用于在CDN出现问题时选择将要使用的文件
asp-fallback-href-exclude	用于在CDN出现问题时选择将被排除的文件
asp-fallback-href-test-class	用于指定测试CDN的CSS类名
asp-fallback-href-test-property	用于指定测试CDN的CSS属性
asp-fallback-href-test-value	用于指定测试CDN的CSS值

1．选择样式表

LinkTagHelper与ScriptTagHelper支持许多相同的特性，包括支持使用通配符选择或排除CSS文件，所以不必单个指定。能够准确选择CSS文件与选择JavaScript文件一样重要，因为样式表文件也有常规和紧缩版本，同样支持源码映射。对于流行的

Bootstrap包，本书一直用它来修饰元素。wwwroot/lib/bootstrap/dist/css文件夹中包含一些样式表文件，如果在Solution Explorer窗格中展开所有的项目，你将看到多个文件，如图25-5所示。

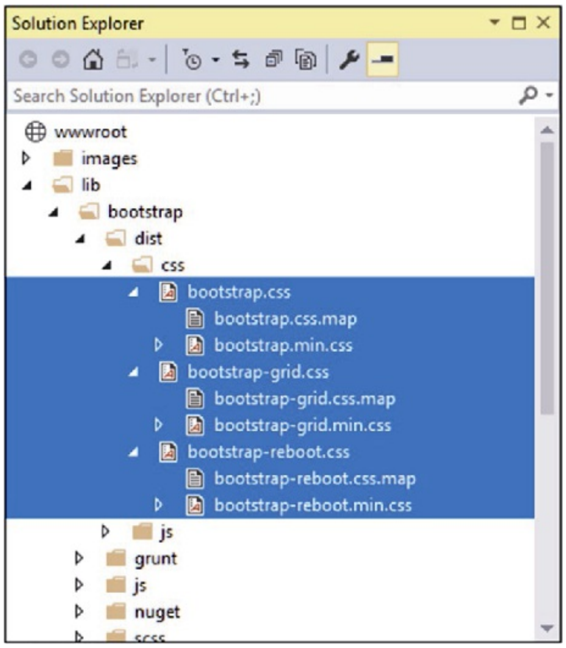


图25-5　多个文件

文件boostrap.css表示常规样式表，文件boostrap.min.css表示紧缩版本，文件bootstrap.css.map表示源码映射，这里的其他文件用于提供特定功能，但本章并不涉及。代码清单25-

11使用link元素的asp-href-include属性来选择紧缩版本的样式表（还删除了加载jQuery的script元素，因为这里已不再需要）。

代码清单25-11　在Views/Shared文件夹下的_Layout.cshtml文件中选择样式表

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link rel="stylesheet"
          asp-href-include="/lib/bootstrap/dist/**/*.mi
n.css"
          asp-href-exclude="**/*-reboot*,**/*-grid*"/>
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

在选择JavaScript文件时同样需要注意细节，因为很容易非你所愿地生成用于选择多个同样文件的不同版本的link元素。选择JavaScript文件有3种方

式：限制通配符，使用asp-href-exclude属性排除文件，使用environment元素在重复的元素集合中进行选择。

2．使用CDN

LinkTag类提供了一系列属性用于在CDN失效时提供到本地内容的回退控制，但测试样式表是否加载的过程相比JavaScript文件有一点复杂。代码清单25-12使用MaxCDN URL来加载Bootstrap库，这仅仅用来展示微软平台之外的另一个选择（MaxCDN是Bootstrap项目推荐的CDN）。

代码清单25-12　在Views/Shared文件夹下的_Layout.cshtml文件中使用CDN加载CSS

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
```

```
    <link href="*****://maxcdn.bootstrapcdn.***/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <div>@RenderBody()</div>
</body>
</html>
```

属性href用于指定CDN地址，这里还使用asp-fallback-href-include属性来选择在CDN不可用时将被使用的文件。为了测试CDN是否工作，需要使用3个不同的属性，并理解所使用的CSS样式表定义的CSS类。

CSS回退特性通过为文档添加meta元素而起作用，其由asp-fallback-test-class属性定义的类添加。以上代码指定了btn类，这意味着如下元素将被添加

到发送到浏览器的HTML中：

```
<meta name="x-stylesheet-fallback-test" class="btn" />
```

指定的CSS类必须在通过CDN加载的样式表中定义。这里指定的btn类为Bootstrap按钮元素提供基本的格式化。

属性asp-fallback-test-property用于指定由CSS类定义的属性，属性asp-fallback-test-value用于指定将被设置的值。标签助手会添加JavaScript到视图中以测试meta元素中的CSS属性值，从而确定样式表是否被加载，并且如果没有加载，就为回退文件添加link元素。Bootstrap的btn类已设置display属性为inline-block，从而测试并查看浏览器是否能从CDN加载Bootstrap样式表。

提　示

要测试第三方库，比如Bootstrap，最简单的方法是按F12键。为检验代码清单25-12中的测试，为按钮指定btn类，然后在浏览器中检查，查看类修改的CSS属性，这比试图阅读长且复杂的样式表要容易。

25.4　使用超链接元素

超链接元素a是在应用程序中进行导航的基本工具，可通过发送GET请求到应用程序来请求不同的内容。AnchorTagHelper类用于转换a元素的href属性以便使用路由系统生成指向的URL地址。该标

签助手支持的属性如表25-5所示。

表**25-5**　用于超链接元素的内置标签助手类定义的属性

属性	描述
asp-action	指定URL指向的操作方法
asp-controller	指定URL指向的控制器
asp-area	指定URL指向的区域
asp-fragment	用于指定URL片段（在#字符之后应用）
asp-host	指定URL指向的主机名
asp-protocol	指定URL使用的协议
asp-route	指定用于生成URL的路由名称
asp-route-*	以asp-route-开头的属性用于指定URL附加值，例如asp-route-id属性用于指定路由系统中id片段的值

AnchorTagHelper简单且可预测，使得使用应用程序的路由系统在a元素中生成URL变得容易。代码清单25-13更新了Index.cshtml文件中的a元素，以便其href属性由标签助手生成。

代码清单25-13　在Views/Home文件夹下的Index.cshtml文件中转换超链接元素

```
@model IEnumerable<City>

@{  Layout = "_Layout"; }

<table class="table table-sm table-bordered">
    <thead class="bg-primary text-white">
        <tr>
            <th>Name</th>
            <th>Country</th>
            <th class="text-right">Population</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var city in Model) {
            <tr>
                <td>@city.Name</td>
                <td>@city.Country</td>
                <td class="text-right">@city.Population
?.ToString("#,###")</td>
            </tr>
        }
```

```
    </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</a>
```

如果运行应用程序并访问/Home/Index，你将看到标签助手将a元素转换为如下形式：

```
<a class="btn btn-primary" href="/Home/Create">Create</a>
```

25.5　使用图像元素

ImageTagHelper类用于通过img元素的src属性为图片提供缓存清除功能，在确保对图片的修改能够立即得到反馈的同时，允许应用程序获得缓存的好处。可使用img元素的asp-append-version属性启用缓存清除功能。

代码清单25-14在共享布局中添加了一个img元素（为了简化，可以重置style元素以便使用本地文

件）。

代码清单25-14　在Views/Shared文件夹下的_Layout.cshtml文件中添加图片

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <img src="/images/city.png" asp-append-version="tru
e" />
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，你将看到图片显示在每个

页面的顶部。如果检查发送到浏览器的HTML，你将看到用于请求图片文件的URL包含了版本校验和，如下所示：

```
<img src="/images/city.png?v=KaMNDSZFbzNpE8Pkb30EXcAJufRcRDpKh0K_IIPNc7E" />
```

与JavaScript和CSS样式表的缓存清除类似，包含在URL中的校验和将保持不变，直到文件被修改为止。

25.6　使用数据缓存

MVC包含的内存缓存用于缓存内容片段以便加速视图的渲染。被缓存的内容使用视图中的cache元素指定，由CacheTagHelper类使用表25-6所示的属性处理。

注　意

缓存是重用内容段落的有用工具，以便它们不需要为每次请求生成一次。但是，使用缓存事实上需要仔细考虑和规划。缓存可以改进应用程序的性能，也会导致奇怪的效果，比如用户收到过时的内容，或者多个缓存包含不一致的版本，等等，因为应用程序缓存的前一个版本的内容与新版本的内容混杂在一起会导致更新部署被中断。除非清晰定义了要解决的性能问题，并确认理解缓存将导致的影响，否则不要启用缓存。

表**25-6**　用于**cache**元素的内置标签助手类定义的属性

属性	描述

enabled	用于控制cache元素的内容是否被缓存，若省略，表示启用缓存
expires-on	用于指定缓存内容的绝对过期时间，表达式为DateTime值
expires-after	用于指定缓存内容过期的相对时间，表达式为TimeSpan值
expires-sliding	用于指定当缓存内容过期时，从最后使用开始的区间，表达式为TimeSpan值
vary-by-header	用于指定请求头的名称，它将用于管理不同版本的缓存内容
vary-by-query	用于指定查询串的键，它将用于管理不同版本的缓存内容
vary-by-route	用于指定路由变量的名称，它将用于管理不同版本的缓存内容
vary-by-cookie	用于指定Cookie的名称，它将用于管理不同版本的缓存内容
vary-by-user	用于指定是否验证用户的名称，它将用于管理不同版本的缓存内容
vary-by	用于提供管理不同版本缓存内容的键值
priority	用于指定相对优先级，在内存缓存耗尽并清理未过期缓存内容时，该属性的值可供参考

为了演示cache属性的操作方式，创建Components文件夹，并添加名为TimeViewComponent.cs的类文件，用它定义代码清单25-15所示的视图组件。

代码清单25-15　Components文件夹下的TimeViewComponent.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Cities.Components {

    public class TimeViewComponent : ViewComponent {

        public IViewComponentResult Invoke() {
            return View(DateTime.Now);
        }
    }
}
```

Invoke方法使用默认视图，并提供DateTime对象作为视图模型。为了给视图组件提供视图，创建Views/Home/Components/Time文件夹，添加名为

Default.cshtml的视图文件，内容如代码清单25-16所示。

代码清单25-16　Views/Home/Components/Time文件夹下的Default.cshtml文件的内容

```
@model DateTime

<div class="m-1 p-1 bg-info text-white">
    Rendered at @Model.ToString("HH:mm:ss")
</div>
```

DateTime模型对象用于显示当前时间，准确到秒。代码清单25-17已将25.5节的img元素替换为@await Component.InvokeAsync表达式以调用视图组件。

代码清单25-17　在Views/Shared文件夹下的_Layout.cshtml文件中使用视图组件

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
```

```
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @await Component.InvokeAsync("Time")
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，你将看到标题栏显示了内容渲染的时间。等待几秒后重新加载页面，你将看到显示的时间发生了变化，如图25-6所示。

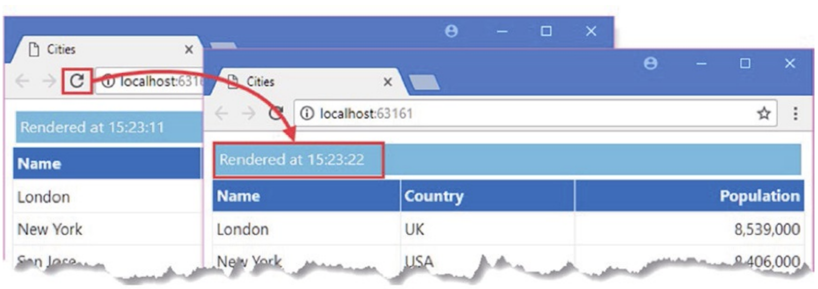


图25-6　在示例应用中显示时间

元素cache用来环绕将被添加到缓存中的内容。代码清单25-18使用cache元素将视图组件添加到了缓存中。

代码清单25-18　在Views/Shared文件夹下的_Layout.cshtml文件中缓存内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    < link href="https://maxcdn.bootstrapcdn.com/bootst
rap/4.0.0-alpha.6/css/bootstrap.min.css"
           asp-fallback-href-include="/lib/bootstrap/di
st/**/*.min.css"
           asp-fallback-href-exclude="**/*-reboot*,**/*
-grid*"
           asp-fallback-test-class="btn"
           asp-fallback-test-property="display"
           asp-fallback-test-value="inline-block"
           rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache>
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
```

```
</html>
```

在没有任何属性的情况下应用cache元素会使MVC重用这些内容以满足所有未来的请求。如果启动应用程序，则会缓存视图组件生成的内容，所以即使页面重新加载也会显示同样的时间。

提 示

CacheTagHelper类使用的缓存是基于内存的，这意味着缓存容量受限于可用的内存。当缓存容量短缺的时候，将从缓存中退出内容，在应用程序停止或重启时，整个内容都将丢失。

25.6.1　设置缓存过期时间

属性expires-*允许指定内容何时过期，使用绝对时间（相对于当前时间而言）或缓存内容不被请求的区间来表示。代码清单25-19使用expires-after属性指定内容应被缓存15s。

代码清单25-19　在Views/Shared文件夹下的_Layout.cshtml文件中设置缓存过期时间

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
```

```
    <cache expires-after="@TimeSpan.FromSeconds(15)">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序，你将看到缓存的数据在15s后过期，重新加载页面后将调用视图组件，并创建另一新的缓存过期时间为15s的条目。

1. 设置固定过期时间

可以使用expires-on属性指定固定的缓存内容过期时间，该属性接收一个DateTime值，如代码清单25-20所示。

代码清单25-20　在Views/Shared文件夹下的_Layout.cshtml中设置固定过期时间

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
```

```
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache expires-on="@DateTime.Parse("2100-01-01")">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

以上代码指定数据将被缓存到2100年。由于应用程序很可能在22世纪开始前重新启动，因此这不是什么有用的缓存策略，但它展示了在内容被缓存的时候，如何指定固定的未来时间，而不是相对于当前时刻的过期时间。

2．设置最后使用缓存期限

属性expires-sliding用于指定如果没有被缓存访问过，多长时间之后内容将过期。代码清单25-21指定过期时间为10s。

代码清单25-21　在Views/Shared文件夹下的_Layout.cshtml文件中指定最后使用缓存期限

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache expires-sliding="@TimeSpan.FromSeconds(10)">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
```

```
</html>
```

通过运行应用程序并定期重新加载页面，就可以看到express-sliding属性的效果。只要在10s内重新加载页面，缓存的内容将被使用。如果等待的时间超过10s并重新加载页面，那么缓存的内容将被丢弃，视图组件将用于生成新的内容，并且此过程将重新开始。

25.6.2 使用缓存变体

默认情况下，所有请求接收相同的缓存内容。CacheTagHelper类可以对缓存的内容维护多种不同版本，并使用它们来满足不同种类的HTTP请求，使用名称以不同方式开头的vary-by属性之一进行设置即可。代码清单25-22展示了如何使用vary-by-route属性来基于路由系统匹配的action名称创建缓存变体。

代码清单25-22　在Views/Shared文件夹下的_Layout.cshtml文件中创建缓存变体

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache expires-sliding="@TimeSpan.FromSeconds(10)"
vary-by-route="action">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

如果运行应用程序并使用两个浏览器窗口来访问/Home/Index和/Home/Create，你将看到每个窗口

接收各自的缓存内容，因为每个请求会处理不同的action路由值。CacheTagHelper类支持不同变体的一系列属性，包括为单个用户缓存内容。

vary-by请求头允许使用任何数据值定义任意缓存变体。代码清单25-23通过指定直接从路由数据中提取的值再现了vary-by-route属性的效果。

代码清单25-23　在Views/Shared文件夹下的_Layout.cshtml文件中指定自定义的缓存变体

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstr
ap/4.0.0-alpha.6/css/bootstrap.min.css"
          asp-fallback-href-include="/lib/bootstrap/dis
t/**/*.min.css"
          asp-fallback-href-exclude="**/*-reboot*,**/*-
grid*"
          asp-fallback-test-class="btn"
          asp-fallback-test-property="display"
          asp-fallback-test-value="inline-block"
          rel="stylesheet" />
```

```
</head>
<body class="m-1 p-1">
    <cache expires-sliding="@TimeSpan.FromSeconds(10)"
           vary-by="@ViewContext.RouteData.Values["acti
on"]">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

属性vary-by可以用于创建更复杂的缓存变体，应该尽量慎用，因为它很容易失控，导致创建的变体如此特殊，以至于缓存的内容在过期前永远不会被重用。

使用应用程序相关的URL

最后介绍的内置标签助手类是UrlResolutionTagHelper，用于对应用程序相关的URL提供支持，它们是以~字符作为前缀的URL。代码清单25-24修改了共享布局中的link元素以便使用显式定义的URL，而不是使用标签助手从路由系

统生成URL。

代码清单25-24　在Views/Shared文件夹下的_Layout.cshtml文件中使用显式定义的URL

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="/lib/bootstrap/dist/css/bootstrap.min.c
ss" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache expires-sliding="@TimeSpan.FromSeconds(10)"
           vary-by="@ViewContext.RouteData.Values["acti
on"]">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

显式的URL是完全可以接受的，但你要明白，如果更新应用程序的URL架构，你将不得不更新它们。

但是，某些应用程序将被部署到共享环境中，单个服务器通过向URL添加前缀来支持多个应用程序。代码清单25-25修改了应用程序的配置，以便将请求管线设置为处理具有mvcpp前缀的请求，从而模拟共享环境。

代码清单25-25　在Cities文件夹下的Startup.cs文件中添加URL前缀

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Cities.Models;

namespace Cities {
    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.Map("/mvcapp", appBuilder => {
                appBuilder.UseStatusCodePages();
                appBuilder.UseDeveloperExceptionPage();
                appBuilder.UseStaticFiles();
                appBuilder.UseMvcWithDefaultRoute();
            });
        }
    }
}
```

Map方法允许使用不同的前缀设置多个请求管线。在日常MVC开发中这通常不是什么有用的功能，因为可以使用路由系统在MVC应用程序内创建URL前缀。但对于本章，这是一个十分有用的特性，因为它意味着被客户端请求的任何URL将包括对静态内容的请求。

通过运行应用程序并访问/mvcapp（该地址现在是应用程序的默认地址，并且指向Home控制器的Index操作方法），你可以查看出现的问题。现在所有的URL都以/mvcapp开始，对于link元素内的样

式表，显式的URL不起作用，这意味着应用程序的内容不能被修饰，如图25-7所示。

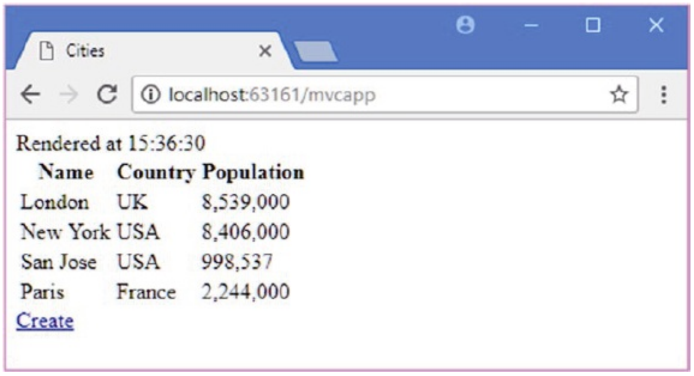


图25-7　显式定义URL的效果

通过在URL中包含这个前缀，你可以解决该问题，但是因为前缀可能在部署时更新或在开发时不知道，这种方案并不可行。更好的方案是使用应用程序相关的URL，这样静态内容的路径就可以表示为相对于已配置的任何前缀，如代码清单25-26所示。

代码清单25-26　在Views/Shared文件夹下的_Layout.cshtml文件中使用应用程序的相对URL

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>Cities</title>
    <link href="~/lib/bootstrap/dist/css/bootstrap.min.
css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    <cache expires-sliding="@TimeSpan.FromSeconds(10)"
           vary-by="@ViewContext.RouteData.Values["acti
on"]">
        @await Component.InvokeAsync("Time")
    </cache>
    <div>@RenderBody()</div>
</body>
</html>
```

~符号由UrlResolutionTagHelper检测，将~符号替换为到达wwwroot文件夹内容所需的路径。如果运行应用程序，你将看到内容被修饰了，检查发送到浏览器的HTML内容，你将看到link元素包含了使用mvcapp前缀的URL：

```
<link href="/mvcapp/lib/bootstrap/dist/css/bootstrap.mi
n.css" rel="stylesheet" />
```

UrlResolutionTaghelper能在各种元素中查询URL，如表25-7所示。

表**25-7**　可由**UrlResolutionTagHelper**转换的元素及其属性

元素	属性
a	href
applet	archive
area	href
audio	src
base	href
blockquote	cite
button	formaction
del	cite
embed	src
form	action

html	manifest
iframe	src
img	src和srcset
input	src和formaction
ins	cite
link	href
menuitem	icon
object	archive和data
q	cite
script	src
source	src和srcset
track	src
video	src和poster

提　示

如果使用另一种内置的标签助手从路由系统生成URL，那么生成的HTML将自动包括任何必要的前缀，这是从HttpRequest.PathBase上下文属性中获取的，并且值由宿主应用程序的服务器提供。

25.7　小结

本章介绍了除了表单标签助手之外的内置标签助手。这些标签助手用于帮助管理对JavaScript文件和CSS样式表文件的访问，为超链接元素创建URL，为图片执行缓存清理，缓存数据，以及转换

与应用程序相关的URL，等等。下一章将介绍模型绑定，用于处理HTTP请求中的数据，以便数据可以在MVC应用程序内部轻松使用。

第26章　模型绑定

模型绑定是使用HTTP请求中的数据来创建.NET对象的过程，以便为操作方法提供需要的参数。本章将描述模型绑定的工作方式，展示模型绑定如何绑定简单类型、复杂类型以及集合，并演示如何控制绑定过程来指定将请求的哪一部分作为操作方法要求的值。表26-1介绍了模型绑定的背景。

表26-1　模型绑定的背景

问题	答案
模型绑定是什么？	模型绑定是使用HTTP请求中的数据来创建.NET对象的过程，以便为操作方法提供需要的参数
模型绑定有何作用？	模型绑定允许操作方法使用C#类型来定义参数，参数可自动从请求中接收数据，而不需要直接检查、解析和处理请求数据
如何使用模	对于最简单的场景，操作方法会定义参数，参数名称用来从HTTP请求中抽取数据。可以通过对操作方法的参数应用特性来配置请求的哪一部分用于数

型绑定？	据抽取
模型绑定有何陷阱或限制？	主要的陷阱是从请求的错误部分获取数据。26.2节将阐述用于搜索请求数据的方式，搜索的位置可以使用特性显式指定
模型绑定有何替代品？	操作方法可以完全不需要声明参数，可以使用上下文对象直接从HTTP请求对象中获取数据。但是，代码更复杂，且难以理解和维护

表26-2列出了本章要介绍的操作。

表**26-2**　本章要介绍的操作

操作	方法	代码清单
绑定简单类型或集合	为操作方法添加参数	代码清单26-1～代码清单26-10和代码清单26-23～代码清单26-29
绑定复杂类型	确认视图生成的HTML是结构良好的	代码清单26-11～代码清单26-19
选择绑定属性	使用Bind特性指定数据值的名称，或者使用BindNever特性从绑定过程中排除模型属性	代码清单26-20～代码清单26-22
指定数据绑定值的	为操作方法的参数或模型属性应用特性，标	代码清单26-30～代码清单26-38

来源	识绑定值应该来自何方	

26.1　准备示例项目

本章将使用ASP.NET Core Web Application (.NET Core)模板创建一个新的名为MvcModels的Empty项目。

26.1.1　创建模型和存储库

这里创建Models文件夹，并在其中创建名为Person.cs的类文件，用它定义代码清单26-1所示的类和枚举。

代码清单26-1　Models文件夹下的Person.cs文件的内容

```
using System;

namespace MvcModels.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
```

```
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }
    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

接着，添加名为Repository.cs的类文件到Models文件夹中以定义接口和实现，如代码清单26-2所示。

代码清单26-2　Models文件夹下的Repository.cs文件的内容

```
using System.Collections.Generic;

namespace MvcModels.Models {
```

```
    public interface IRepository {
        IEnumerable<Person> People { get; }

        Person this[int id] { get; set; }
    }

    public class MemoryRepository : IRepository {
        private Dictionary<int, Person> people
                    = new Dictionary<int, Person> {
            [1] = new Person {PersonId = 1, FirstName =
 "Bob",
                LastName = "Smith", Role = Role.Admin},
            [2] = new Person {PersonId = 2, FirstName =
 "Anne",
                LastName = "Douglas", Role = Role.User}
,
            [3] = new Person {PersonId = 3, FirstName =
 "Joe",
                LastName = "Able", Role = Role.User},
            [4] = new Person {PersonId = 4, FirstName =
 "Mary",
                LastName = "Peters", Role = Role.Guest}
        };

        public IEnumerable<Person> People => people.Val
ues;

        public Person this[int id] {
            get {
                return people.ContainsKey(id) ? people[
id] : null;
            }
            set {
                people[id] = value;
            }
        }
```

```
    }
}
```

IRepository接口定义了People属性来获取模型中的所有对象，还定义了索引器以允许获取或存储单个对象。MemoryRepository类使用字典实现这个接口并提供了一些默认内容。这个存储库实现是非持久的，所以在应用停止或重新启动的时候，应用的状态将被重置。

26.1.2　创建控制器和视图

这里创建Controllers文件夹，添加名为HomeController.cs的类文件，并用它定义代码清单26-3所示的控制器。Home控制器基于依赖注入来接收存储库，在Index方法中，通过PersonId属性可在存储库中获取单个Person对象。

代码清单26-3　Controllers文件夹下的HomeController.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index(int id) => View(reposit
ory[id]);
    }
}
```

为了给操作方法提供视图，这里创建Views/Home文件夹，使用代码清单26-4所示的内容在其中添加名为Index.cshtml的Razor文件，将模型对象的一些属性展示在表格中。

代码清单26-4　Views/Home文件夹下的Index.cshtml文件的内容

```
@model Person
@{ Layout = "_Layout"; }
```

```
<div class="bg-primary m-1 p-1 text-white"><h2>Person</
h2></div>

<table class="table table-sm table-bordered table-strip
ed">
    <tr><th>PersonId:</th><td>@Model.PersonId</td></tr>
    <tr><th>First Name:</th><td>@Model.FirstName</td></
tr>
    <tr><th>Last Name:</th><td>@Model.LastName</td></tr
>
    <tr><th>Role:</th><td>@Model.Role</td></tr>
</table>
```

Index视图基于共享布局。创建Views/Shared文件夹，在其中添加名为_Layout.cshtml的布局文件，内容如代码清单26-5所示。

代码清单26-5　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"
/>
    <title>@ViewBag.Title</title>
    <link asp-href-include="/lib/bootstrap/dist/**/*.mi
n.css" rel="stylesheet" />
    @RenderSection("scripts", false)
</head>
```

```
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

以上布局包含用于Bootstrap样式表的link元素并且渲染视图内容，还包含可选的scripts部分（section），本章的后面将会用到。为了简化本章用到的视图，在Views文件夹的_ViewImports.cshtml文件中添加包含模型类的命名空间，如代码清单26-6所示。

代码清单26-6　在Views文件夹下的_ViewImports.cshtml文件中导入命名空间

```
@using MvcModels.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

视图基于Bootstrap CSS框架。下面使用Bower Configuration File模板，在项目的根目录中创建bower.json文件，将Bootstrap添加到项目中。添加的包如代码清单26-7所示。

代码清单26-7　在MvcModels文件夹下的bower.json文件中添加Bootstrap包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

26.1.3　配置应用

为了初始化示例应用，在Startup类中启用MVC框架和开发中用到的其他中间件，如代码清单26-8所示。这里还为存储库创建了服务以便控制器可以访问数据模型。

代码清单26-8　MvcModels文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using MvcModels.Models;
namespace MvcModels {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<IRepository, MemoryRe
pository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{actio
n=Index}/{id?}");
            });
        }
    }
}
```

启动应用并访问/Home/Index/1，将生成图26-1所示的结果。现在，默认的URL地址将导致错误。

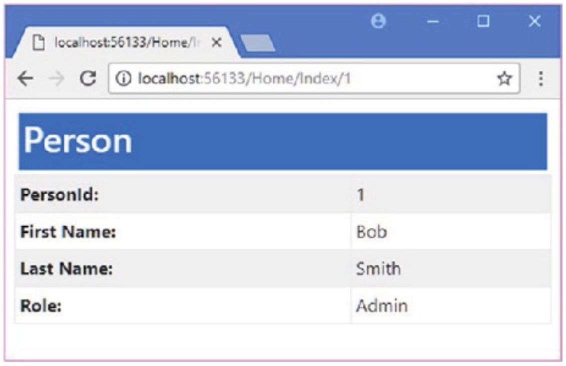


图26-1　运行示例应用

26.2　理解模型绑定

模型绑定是HTTP请求与C#操作方法之间优雅的桥梁。多数MVC应用在某种程度上依赖模型绑定，包括本章的示例应用。前面在测试示例应用的时候，模型绑定就被应用了。请求的URL中包含了希望查阅的Person对象的PersonId属性值，如下所示：

```
/Home/Index/1
```

MVC对这一部分URL进行转换，并且在调用Home控制器的Index方法以服务请求的时候将之作

为参数。

```
...
public ViewResult Index(int id) => View(repository[id])
;
...
```

为了调用Index方法，MVC需要一个值作为id参数，提供这个值就是模型绑定系统的责任，模型绑定系统负责为调用操作方法提供数据。

模型绑定基于模型绑定器，模型绑定器的职责是从请求的某个部分或应用本身提供数据。默认的模型绑定从3个方面提供数据：

- 表单数据；
- 路由变量；
- 查询串。

每个数据源按照顺序被检查，直到参数的值被发现。示例应用中没有表单数据，所以也就不用从

表单中进行查找。但是，应用的配置中包含名为id的路由片段，因而允许模型绑定系统提供用于调用Index方法的值。搜索在发现合适的值之后就会停止，这意味着查询串不会用来搜索数据。

提　示

26.3节将说明如何使用特性来指定模型绑定的数据源。这将允许从指定的数据源获取数据，例如查询串，即使表单或路由数据中存在合适的数据。

知道数据的搜索顺序很重要，因为请求中可能包含多个值，比如下面这个URL：

```
/Home/Index/3?id=1
```

路由系统将处理请求并匹配URL模板中的id片段为3，查询串中包含id为1的值。由于路由系统在查询串之前搜索数据，Index方法将收到值3，查询串的值将被忽略。

另外，如果请求URL中没有id片段，查询串将会被处理，这意味着这样的URL也允许模型绑定系统为MVC提供id值，以便调用Index方法。

```
/Home/Index?id=1
```

图26-2显示了这两种URL的结果。

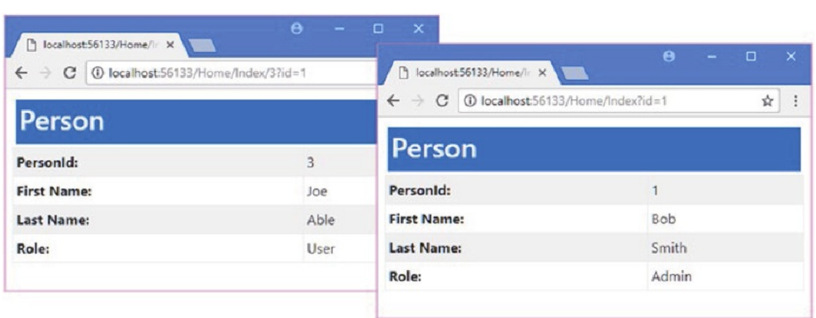


图26-2　数据源次序对模型绑定的影响

26.2.1　默认绑定值

模型绑定是尽力而为的特性，这意味着MVC将使用模型绑定来试着获取调用操作方法所需的值，但是在值不能提供时也仍然调用操作方法，这可能导致一些意想不到的行为。例如，访问/Home/Index将导致图26-3所示的异常。

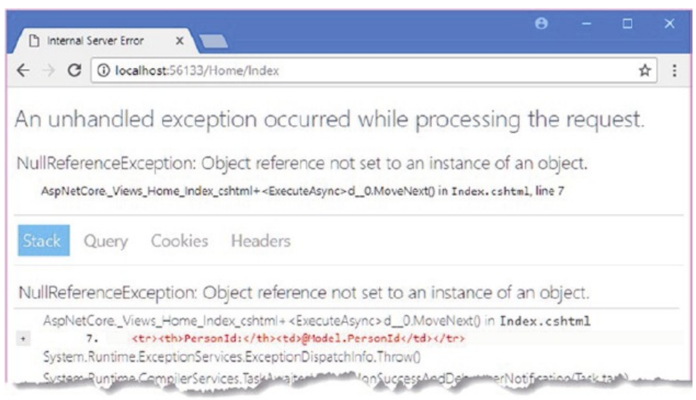


图26-3　异常

图26-3所示的异常不是由模型绑定系统报告的。当处理由Index方法选择的Index视图时，该异常出现。为了调用Index方法，MVC不得不为id参数提供值，所以MVC请求每个模型绑定器检查各自

的部分来提供值。

示例中没有表单数据，没有id路由片段，URL中也没有查询串，这意味着模型绑定不能提供值，MVC必须为id参数提供某些值来调用Index方法，MVC使用了默认值并期望这是最好的。对于int类型参数，默认值是0，这就是导致异常的原因。在Index方法的定义中，使用id参数的值从存储库中获取模型对象。

```
...
public ViewResult Index(int id) => View(repository[id])
;
...
```

当MVC使用默认值时，操作方法试图使用id参数值0来获取数据对象。没有这样的对象，存储库返回null，这个结果随后被传递给控制器的View方法以为Index.cshtml文件提供视图模型对象。当Index.cshtml文件中的Razor表达式试图访问视图模

型对象的属性时，就会导致图26-3所示的NullReferenceException异常。

这意味着操作方法必须应对模型绑定系统提供的默认值。这可以通过多种方式来完成。可以添加默认值到 URL 路由模板中，为操作方法提供默认参数，或者确保操作方法不会将错误值作为一部分传递给响应。最好的处理方式取决于操作方法做什么：代码清单 26-9 采用了最后一种方式，修改操作方法以确保一个Person对象总是能传递给View方法，即使id参数并不对应数据模型中的对象。

代码清单26-9　在Controllers文件夹下的HomeController.cs文件中防范默认的模型绑定值

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Linq;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;
```

```
        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index(int id) =>
            View(repository[id] ?? repository.People.Fi
rst());
    }
}
```

当使用id参数的值不能取回对象的时候，操作方法使用LINQ和null合并操作符来返回存储库中的第一个对象。

26.2.2 绑定简单值

当有合适的值可用时，该值必须被转换为C#值，以便可以用于调用操作方法。使用请求中的简单类型可以从字符串中解析得到数据项，包括数值、布尔值、日期以及字符串值。

Index方法的id参数为int类型，所以模型绑定过程通过将id片段解析为整数值提供给MVC。

如果请求值不能被转换（例如，提供apple给需要int类型值的参数），模型绑定系统将不能为应用程序提供值，默认值将会被使用。

这会出现问题，因为在两种情况下操作方法都将收到默认值0。第一种情况是，请求中包含不能被解析为参数类型的值，如URL为/Home/Index/Apple。第二种情况是，请求中包含可以解析的值但碰巧是0，如URL为/Home/Index/0。

大多数应用程序需要区分这些状况，最简单的方式是使用可空类型作为操作方法的参数，如代码清单26-10所示。

代码清单26-10　在Controllers文件夹下的HomeController.cs文件中使用可空类型

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
```

```
namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
    }
}
```

默认可空类型的值是null，这允许区分请求中不包括可以解析为整数值的情况和值碰巧是0的情况。如果可空参数没有值或者值没有关联模型中的对象，那么示例中Index方法的实现将使用NotFound方法来返回404错误。这是一种比简单地期望模型中的第一个对象十分适当的更稳健的方

式，也是之前采用的方式。

26.2.3　绑定复杂类型

当操作方法的参数是复杂类型的时候（换句话说，任何不能从单个字符串值解析到的类型），模型绑定过程将使用反射来获取目标类型的公共属性集合，然后依次对每个属性执行绑定过程。为了演示这是如何工作的，在Home控制器中添加两个操作方法，如代码清单26-11所示。

代码清单26-11　在Controllers文件夹下的HomeController.cs中添加两个操作方法

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }
```

```
        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);

        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);
    }
}
```

没有参数的Create方法版本会创建一个新的Person对象并传递给View方法，View方法将影响选择关联到操作方法的默认视图。在Views/Home文件夹中创建名为Create.cshtml的视图文件，并添加代码清单26-12所示的标记。

代码清单26-12　Views/Home文件夹下的Create.cshtml文件的内容

```
@model Person
@{
    ViewBag.Title = "Create Person";
    Layout = "_Layout";
}

<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="PersonId"></label>
        <input asp-for="PersonId" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" class="form-control"
 />
    </div>
    <div class="form-group">
        <label asp-for="LastName"></label>
        <input asp-for="LastName" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="Role"></label>
        <select asp-for="Role" class="form-control"
                asp-items="@new SelectList(Enum.GetName
s(typeof(Role)))"></select>
    </div>
    <button type="submit" class="btn btn-primary">Submi
t</button>
</form>
```

Create视图包含一个将数据回送给Home控制器

中使用HttpPost特性修饰的Create方法的表单元素，可通过它为Person对象的某些属性提供值。

操作方法接收表单数据，然后使用/Views/Home/Index.cshtml文件显示它们。启动应用，导航到/Home/Create，填写表单，然后单击Submit按钮来看看它们是如何工作的，如图26-4所示。

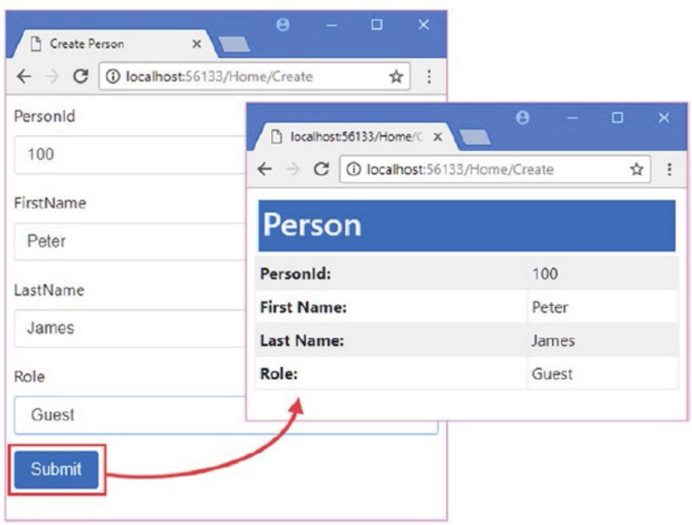


图26-4　填写表单并单击Submit按钮

在把表单数据发送给服务器后，模型绑定系统发现操作方法需要一个Person对象。分析Person类及其公共属性。对于每个简单类型，模型绑定器试

图查找请求值，就像前一个示例那样。

对于这个示例，模型绑定器发现了PersonId属性，并寻找PersonId值，由于表单中包含合适的值——在input元素的设置中使用了asp-for属性的标签助手，因此这个值将被使用。

如果属性为另一复杂类型，处理过程将在新的类型上重复进行。获取公共属性集合，然后模型绑定器试图寻找所有属性的值。不同的是，属性名称是嵌套的。例如，Person类的HomeAddress属性为Address类型，如下所示：

```
using System;

namespace MvcModels.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
```

```
        public Role Role { get; set; }
    }

    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

当寻找Line1属性的值时，模型绑定器为HomeAddress.Line1查找值——由模型对象属性的名称合并嵌套的模型类型属性名组成。

1．创建易于绑定的HTML

前缀意味着视图必须包含模型绑定器查找的信息。使用标签助手可以很容易实现，标签助手能自动将需要的前缀添加到它们想要转换的元素上。代

码清单26-13扩展了表单以便可以获取地址数据。

代码清单26-13　在Views/Home文件夹下更新Create.cshtml文件中的表单

```
@model Person
@{
    ViewBag.Title = "Create Person";
    Layout = "_Layout";
}

<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="PersonId"></label>
        <input asp-for="PersonId" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" class="form-control"
 />
    </div>
    <div class="form-group">
        <label asp-for="LastName"></label>
        <input asp-for="LastName" class="form-control"
/>
    </div>
    <div class="form-group">
        <label asp-for="Role"></label>
        <select asp-for="Role" class="form-control"
                asp-items="@new SelectList(Enum.GetName
s(typeof(Role)))"></select>
```

```
    </div>
    <div class="form-group">
        <label asp-for="HomeAddress.City"></label>
        <input asp-for="HomeAddress.City" class="form-c
ontrol" />
    </div>
    <div class="form-group">
        <label asp-for="HomeAddress.Country"></label>
        <input asp-for="HomeAddress.Country" class="for
m-control" />
    </div>
    <button type="submit" class="btn btn-primary">Submi
t</button>
</form>
```

当使用标签助手的时候，嵌套的属性名称使用C#约定以便外包和内嵌的属性名称以句点分隔——HomeAddress.Country。如果运行应用程序，访问/Home/Create，然后检查发送给浏览器的HTML，你将看到对于一些属性使用了不一样的约定：

```
<div class="form-group">
    <label for="HomeAddress_City">City</label>
    <input class="form-control" type="text" id="HomeAdd
ress_City"
        name="HomeAddress.City" value="" />
</div>
```

```
<div class="form-group">
    <label for="HomeAddress_Country">Country</label>
    <input class="form-control" type="text" id="HomeAddress_Country"
        name="HomeAddress.Country" value="" />
</div>
```

input元素的name属性遵循C#风格，但是label元素的for属性和input元素的id属性使用下画线分隔属性名称。如果希望不使用标签助手来定义HTML元素，那么需要确保使用了同样的命名模式。

作为这一特性的结果，不需要采用任何特殊的行动就可以确保模型绑定器会为HomeAddress属性创建Address对象。在数据可以通过表单提交之后，可以通过编辑Index.cshtml视图文件来显示HomeAddress属性，如代码清单26-14所示。

代码清单26-14　在Views/Home文件夹下的Index.cshtml文件中显示HomeAddress属性

```
@model Person
@{ Layout = "_Layout"; }
```

```
<div class="bg-primary m-1 p-1 text-white"><h2>Person</
h2></div>

<table class="table table-sm table-bordered table-strip
ed">
    <tr><th>PersonId:</th><td>@Model.PersonId</td></tr>
    <tr><th>First Name:</th><td>@Model.FirstName</td></
tr>
    <tr><th>Last Name:</th><td>@Model.LastName</td></tr
>
    <tr><th>Role:</th><td>@Model.Role</td></tr>
    <tr><th>City:</th><td>@Model.HomeAddress?.City</td>
</tr>
    <tr><th>Country:</th><td>@Model.HomeAddress?.Countr
y</td></tr>
</table>
```

如果启动应用并导航到URL地址/Home/Create，就可以为City和Country文本框输入值，通过提交表单可以检查它们是否已被绑定到模型对象，如图26-5所示。

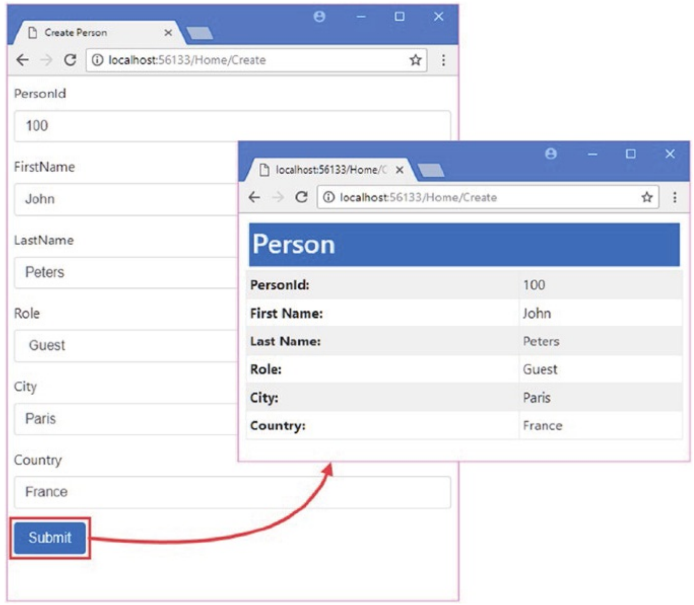


图26-5　绑定复杂对象中的属性

2．指定自定义前缀

在有的场合中，生成的HTML已关联一种类型，但是希望将HTML绑定到另外一种类型。这意味着视图包含的前缀不对应模型绑定期望的结构，所以数据不会被正确处理。为了说明这个问题，添加一个名为AddressSummary.cs的类文件到Models文件夹中，用它定义代码清单26-15所示的类。

代码清单26-15　Models文件夹下的AddressSummary.cs文件的

内容

```
namespace MvcModels.Models {

    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

在Home控制器中添加一个新的使用AddressSummary类的操作方法，如代码清单26-16所示。

代码清单26-16　在Controllers文件夹下的HomeController.cs文件中添加一个操作方法

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }
```

```
        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);

        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);

        public ViewResult DisplaySummary(AddressSummary
 summary) => View(summary);
    }
}
```

新的操作方法名为DisplaySummary，它的AddressSummary参数被传递给View方法以便能被默认视图显示。在/Views/Home文件夹中创建DisplaySummary.cshtml文件，添加代码清单26-17所示的标记内容。

代码清单26-17　Views/Home文件夹下的

DisplaySummary.cshtml文件的内容

```
@model AddressSummary
@{
    ViewBag.Title = "DisplaySummary";
    Layout = "_Layout";
}

<div class="bg-primary m-1 p-1 text-white"><h2>Address<
/h2></div>

<table class="table table-sm table-bordered table-strip
ed">
    <tr><th>City:</th><td>@Model.City</td></tr>
    <tr><th>Country:</th><td>@Model.Country</td></tr>
</table>
```

DisplaySummary视图显示了AddressSummary类中定义的两个属性的值。为了演示使用前缀绑定到不同的模型类型时引发的问题，修改Create.cshtml视图文件中的表单元素以发送数据给DisplaySummary操作，如代码清单26-18所示。

代码清单26-18　在Views/Home文件夹下的Create.cshtml文件中修改表单的操作目标

```
@model Person
```

```
@{
    ViewBag.Title = "Create Person";
    Layout = "_Layout";
}

<form asp-action="DisplaySummary" method="post">

    <!-- HTML elements omitted for brevity -->

</form>
```

启动应用并导航到URL地址/Home/Create，看看发生了什么。提交表单后，你为City和Country文本框输入的值不会显示在DisplaySummary视图生成的HTML中。

问题在于form元素的name属性有HomeAddress前缀，在试图绑定到AddressSummary类型时，它不是模型绑定器想要寻找的内容。

为解决这个问题，Bind特性可以被应用于操作方法的参数，从而在模型绑定中指定前缀，如代码清单26-19所示。

代码清单26-19　在Controllers文件夹下的HomeController.cs中变更模型绑定前缀

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);
        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);

        public ViewResult DisplaySummary(
            [Bind(Prefix = nameof(Person.HomeAddress))]
 AddressSummary summary)
```

```
                        => View(summary);
    }
}
```

语法虽然笨拙，但是效果非常好。在填充AddressSummary对象的属性时，模型绑定器将在请求中寻找HomeAddress.City和HomeAddress.Country的值。如果运行应用并重新提交表单，你将看City和Country文本框中的值现在正确显示，如图26-6所示。对于简单问题来说，这看起来有些复杂，但是绑定到不同类型的对象很常见，并且是值得学习的技术。

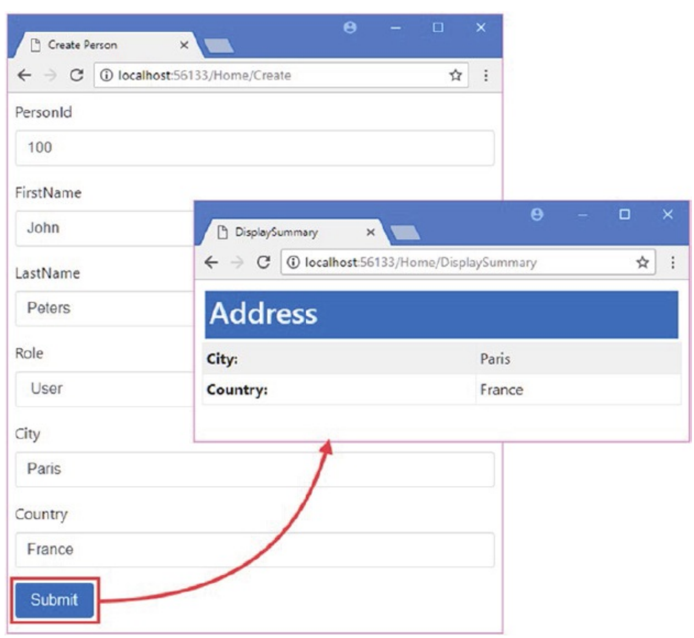


图26-6　绑定到不同类型对象的属性

3．选择性绑定属性

想象一下，AddressSummary类的Country属性特别敏感，用户不应该能够为它指定值。你可以做的第一件事，就是通过确保不会在应用的任何涉及这个属性的视图的HTML元素中包含它，以防止用户查看或编辑这个属性。

然而，恶意用户可以简单地在提交表单数据的时候，编辑发送给服务器的表单数据并挑选合适的Country属性值。这里需要告诉模型绑定器不要从请求中为Country属性绑定值。这可以通过配置操作方法参数的Bind特性来实现，如代码清单26-20所示。

代码清单26-20　在Controllers文件夹下的HomeController.cs文件中指定属性

```
using Microsoft.AspNetCore.Mvc;
```

```
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);

        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);

        public ViewResult DisplaySummary(
            [Bind(nameof(AddressSummary.City), Prefix =
 nameof(Person.HomeAddress))]
                AddressSummary summary) => View(summary
);
    }
}
```

Bind特性的第一个参数是以逗号分隔的应当被模型绑定过程包含的属性名称列表。在这个列表中，指定了City属性应当包含在过程中，由于Country没有列入，这意味着Country属性将被排除。

如果运行应用，访问/Home/Create，然后填充表单并提交，你将看到 Country 属性的值没有显示出来，即使已作为浏览器HTTP POST请求的一部分被发送，如图26-7所示。

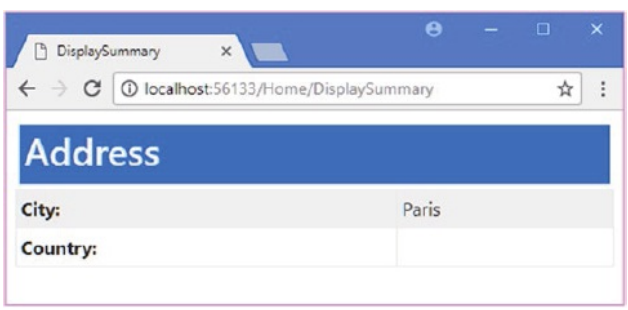


图26-7　在模型绑定过程中排除属性

当Bind特性被应用于操作方法的参数时，它仅仅影响为这个类的实例绑定的操作方法；其他所有

的操作方法将继续试图绑定参数类型定义的所有属性。为了使Bind特性发挥更广泛的影响，可以将Bind特性应用于模型类本身，如代码清单26-21所示。

代码清单26-21　在Models文件夹下的AddressSummary.cs文件中应用Bind特性

```
using Microsoft.AspNetCore.Mvc;

namespace MvcModels.Models {

    [Bind(nameof(City))]
    public class AddressSummary {
        public string City { get; set; }

        public string Country { get; set; }
    }
}
```

也可以使用BindNever特性来显式地排除属性，如代码清单26-22所示，虽然这意味着新添加到模型类的属性将被包含在模型绑定过程中，除非为它们应用这个特性。

代码清单26-22　在Models文件夹下的AddressSummary.cs文件中应用BindNever特性

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace MvcModels.Models {

    public class AddressSummary {
        public string City { get; set; }

        [BindNever]
        public string Country { get; set; }
    }
}
```

提　示

BindRequired特性用于告诉模型绑定过程，请求中必须包含属性的值。如果请求中没有包含必要的值，将会导致模型验证错误，详见第27章。

26.2.4 绑定数组和集合

模型绑定有一些不错的特性用于绑定请求数据到数组和集合，下面进行说明。

1．绑定到数组

模型绑定默认的优雅特性在于支持数组类型的操作方法参数。为了说明这一点，在Home控制器中添加名为Names的新方法，如代码清单26-23所示。

代码清单26-23　在Controllers文件夹下的HomeController.cs文件中添加Names方法

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
```

```
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);
        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);

        public ViewResult DisplaySummary(
            [Bind(nameof(AddressSummary.City), Prefix =
 nameof(Person.HomeAddress))]
                AddressSummary summary) => View(summary
);

        public ViewResult Names(string[] names) => View
(names ?? new string[0]);
    }
}
```

Names方法有一个名为names的字符串数组参

数。模型绑定器将查询名为names且包含这些值的任何数据项来创建数组，为了给这个操作方法提供视图，在Views/Home文件夹中创建名为Names.cshtml的Razor文件，并添加代码清单26-24所示的标记内容。

代码清单26-24　Views/Home文件夹下的Names.cshtml文件的内容

```
@model string[]
@{
    ViewBag.Title = "Names";
    Layout = "_Layout";
}

@if (Model.Length == 0) {
    <form asp-action="Names" method="post">
        @for (int i = 0; i < 3; i++) {
            <div class="form-group">
                <label>Name @(i + 1):</label>
                <input id="names" name="names" class="f
orm-control" />
            </div>
        }
        <button type="submit" class="btn btn-primary">S
ubmit</button>
    </form>
} else {
```

```
    <table class="table table-sm table-bordered table-s
triped">
        @foreach (string name in Model) {
            <tr><th>Name:</th><td>@name</td></tr>
        }
    </table>
    <a asp-action="Names" class="btn btn-primary">Back<
/a>
}
```

Names视图基于视图模型中包含的数据项的个数展示不同的内容。如果没有数据项，该视图将显示包含3个完全相同的input元素的表单，如下所示：

```
...
<form method="post" action="/Home/Names">
    <div class="form-group">
        <label>Name 1:</label>
        <input id="names" name="names" class="form-cont
rol" />
    </div>
    <div class="form-group">
        <label>Name 2:</label>
        <input id="names" name="names" class="form-cont
rol" />
    </div>
    <div class="form-group">
        <label>Name 3:</label>
        <input id="names" name="names" class="form-cont
```

```
rol" />
    </div>
    <button type="submit" class="btn btn-primary">Submi
t</button>
</form>
...
```

当表单被提交之后，模型绑定过程查看目标操作方法并得到一个数组，查询与操作方法参数具有相同名称的数据项。对于这个示例来说，这意味着所有name属性为names的input元素的值将被收集在一起以创建数组，并作为参数调用操作方法。为了查看效果，启动应用，导航到/Home/Names，填充表单并提交，你将看到输入的所有值都显示出来，如图26-8所示。

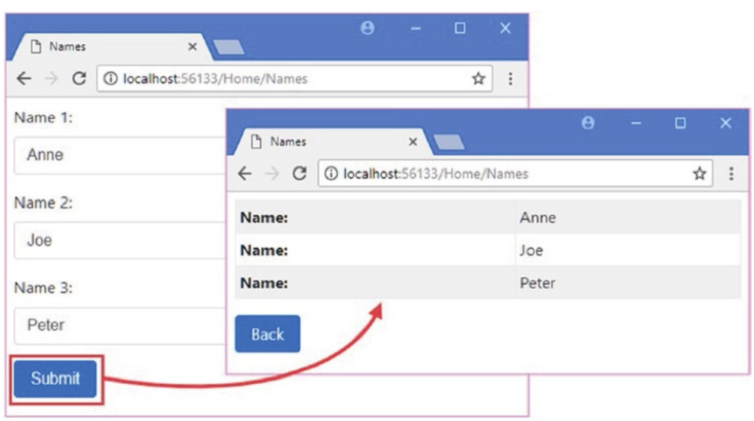


图26-8　绑定到数组

2．绑定到集合

模型绑定器不仅可以绑定数组，还可以绑定集合。代码清单26-25将Names方法的参数类型修改为强类型集合。

代码清单26-25　在Controllers文件夹下的HomeController.cs文件中使用强类型集合

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Collections.Generic;

namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        // ...other action methods omitted for brevity.
..

        public ViewResult Names(IList<string> names) =>
            View(names ?? new List<string>());
```

```
    }
}
```

这里使用了IList< T >接口。不需要指定具体的实现类，虽然根据个人意愿也可以这么做。代码清单26-26修改了Names.cshtml视图文件以使用新的模型类型。

代码清单26-26　在Views/Home文件夹下的Names.cshtml文件中使用集合作为模型类型

```
@model IList<string>
@{
    ViewBag.Title = "Names";
    Layout = "_Layout";
}

@if (Model.Count == 0) {
    <form asp-action="Names" method="post">
        @for (int i = 0; i < 3; i++) {
            <div class="form-group">
                <label>Name @(i + 1):</label>
                <input id="names" name="names" class="f
orm-control" />
            </div>
        }
        <button type="submit" class="btn btn-primary">S
ubmit</button>
    </form>
```

```
} else {
    <table class="table table-sm table-bordered table-s
triped">
        @foreach (string name in Model) {
            <tr><th>Name:</th><td>@name</td></tr>
        }
    </table>
    <a asp-action="Names" class="btn btn-primary">Back<
/a>
}
```

Names方法的功能没有变化，但是现在可以使用集合而不是数组进行工作。

3．绑定到复杂类型的集合

单个数据值可以绑定到复杂类型的集合，这允许在单个请求中收集多个对象（比如示例中的AddressSummary模型类）。代码清单26-27在Home控制器中添加了名为Address的操作方法，参数是一个AddressSummary对象列表。

代码清单26-27　在Controllers文件夹下的HomeController.cs文件中定义操作方法

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Collections.Generic;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Create() => View(new Person()
);

        [HttpPost]
        public ViewResult Create(Person model) => View(
"Index", model);

        public ViewResult DisplaySummary(
            [Bind(nameof(AddressSummary.City), Prefix =
 nameof(Person.HomeAddress))]
                AddressSummary summary) => View(summary
);
```

```
        public ViewResult Names(IList<string> names) =>
            View(names ?? new List<string>());

        public ViewResult Address(IList<AddressSummary>
 addresses) =>
            View(addresses ?? new List<AddressSummary>(
));
    }
}
```

为了给这个新的操作方法提供视图，在Views/Home文件夹中添加名为Address.cshtml的文件，并添加代码清单26-28所示的标记内容。

代码清单26-28　Views/Home文件夹下的Address.cshtml文件的内容

```
@model IList<AddressSummary>
@{
    ViewBag.Title = "Address";
    Layout = "_Layout";
}

@if (Model.Count() == 0) {
    <form asp-action="Address" method="post">
        @for (int i = 0; i < 3; i++) {
            <fieldset class="form-group">
                <legend>Address @(i + 1)</legend>
                <div class="form-group">
```

```
                    <label>City:</label>
                    <input name="[@i].City" class="form
-control" />
                </div>
                <div class="form-group">
                    <label>Country:</label>
                    <input name="[@i].Country" class="f
orm-control" />
                </div>
            </fieldset>
        }
        <button type="submit" class="btn btn-primary">S
ubmit</button>
    </form>
} else {
    <table class="table table-sm table-bordered table-s
triped">
        <tr><th>City</th><th>Country</th></tr>
        @foreach (var address in Model) {
            <tr><td>@address.City</td><td>@address.Coun
try</td></tr>
        }
    </table>
    <a asp-action="Address" class="btn btn-primary">Bac
k</a>
}
```

如果模型集合中没有条目，Address视图将生成一个form元素。这个form元素包含name属性带有数组索引前缀的input元素对，如下所示：

```
...
<form method="post" action="/Home/Address">
    <fieldset class="form-group">
        <legend>Address 1</legend>
        <div class="form-group">
            <label>City:</label>
            <input name="[0].City" class="form-control"
 />
        </div>
        <div class="form-group">
            <label>Country:</label>
            <input name="[0].Country" class="form-contr
ol" />
        </div>
    </fieldset>
    <fieldset class="form-group">
        <legend>Address 2</legend>
        <div class="form-group">
            <label>City:</label>
            <input name="[1].City" class="form-control"
 />
        </div>
        <div class="form-group">
            <label>Country:</label>
            <input name="[1].Country" class="form-contr
ol" />
        </div>
    </fieldset>
    <fieldset class="form-group">
        <legend>Address 3</legend>
        <div class="form-group">
            <label>City:</label>
            <input name="[2].City" class="form-control"
 />
        </div>
```

```
        <div class="form-group">
            <label>Country:</label>
            <input name="[2].Country" class="form-contr
ol" />
        </div>
    </fieldset>
    <button type="submit" class="btn btn-primary">Submi
t</button>
</form>
...
```

表单被提交后，模型绑定器意识到需要创建AddressSummary对象集合，使用name属性的数组索引前缀来获取对象的属性值。属性前缀[0]用于第一个AddressSummary对象，属性前缀[1]用于第二个AddressSummary对象，以此类推。

Address视图为三个索引的对象定义了input元素，在模型集合含有条目的时候显示它们。在加以说明之前，需要从AddressSummary模型类中删除BindNever特性，如代码清单26-29所示；否则，模型绑定器将会忽略Country属性。

代码清单26-29　在Models文件夹下的AddressSummary.cs文件中删除BindNever特性

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace MvcModels.Models {

    public class AddressSummary {

        public string City { get; set; }

        //[BindNever]
        public string Country { get; set; }
    }
}
```

通过启动应用并导航到URL地址/Home/Address，单击Submit按钮发送表单到服务器，就可以看到自定义对象集合的绑定过程是如何工作的。

模型绑定过程将寻找并处理索引的数据值，然后使用它们创建AddressSummary对象集合并提供给操作方法，最后使用View便捷方法将它们传递回视

图以便可以显示出来，如图26-9所示。

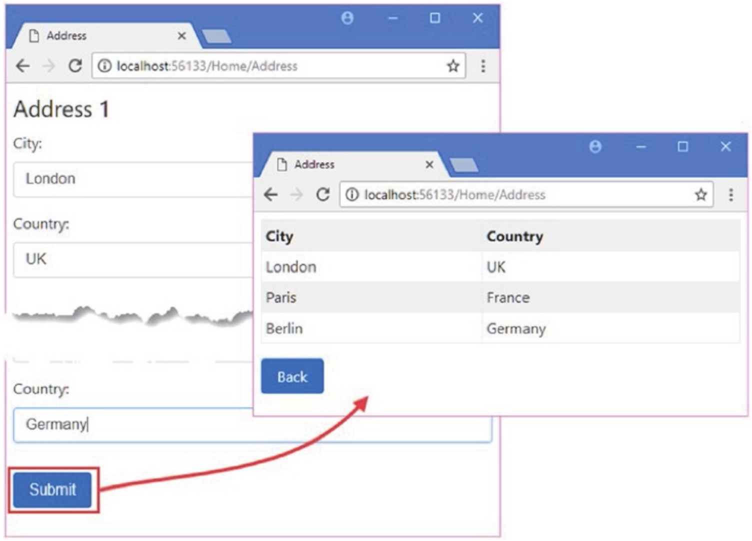


图26-9　绑定到自定义对象的集合

26.3　指定模型绑定源

模型绑定过程默认从3个位置（表单数据、路由变量以及查询串）寻找数据。

默认的搜索顺序并不总是有用，可能是因为希望数据来自请求的某个特定部分，也可能因为希望使用默认不搜索的数据源。模型绑定源包含一系列用来覆盖默认搜索行为的特性，如表26-3所示。

表**26-3**　模型绑定源的特性

特性	描述
FromForm	指定表单数据作为数据绑定源。参数名称默认用于定位表单值，但可以使用Name属性指定不同的名称
FromRoute	指定路由变量作为数据绑定源。参数名称默认用于定位路由变量值，但可以使用Name属性指定不同的名称
FromQuery	指定查询串作为数据绑定源。参数名称默认用于定位查询串值，但可以使用Name属性指定不同的名称
FromHeader	指定请求头作为数据绑定源。参数名称默认用于定位请求头的名称，但可以使用Name属性指定不同的名称
FromBody	指定请求体作为数据绑定源。如果需要从不是表单编码的请求中接收数据（例如在API控制器中，就需要使用这个特性）

26.3.1　选择标准绑定源

FromForm、FromRoute以及FromQuery特性允许指定模型绑定数据将从标准位置获取，但不使用正常的搜索顺序。本章开头使用的是下面这个

URL：

```
/Home/Index/3?id=1
```

这个URL包含可以用于Home控制器中Index方法的id参数的两个可能值。路由系统将URL的最后部分赋予名为id的变量，id变量定义在Startup类中的URL模板中，查询串也包含id值。使用默认的搜索模式意味着模型绑定数据将从路由变量中获取，查询串将被忽略。

为了改变这个行为，代码清单26-30将FromQuery特性应用于Index方法。为了保持示例简单，这里还将前面定义的其他操作方法删除了。

代码清单26-30　在Controllers文件夹下的HomeController.cs中使用查询串

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
```

```
    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index([FromQuery] int? id)
 {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
    }
}
```

以上代码为id参数应用了FromQuery特性，这意味着在模型绑定过程中，查询id值的时候仅仅查询串被使用。

提　示

在指定模型绑定源（比如查询串）时，仍然可以绑定复杂类型。对于参数类型的每个简单属性，模型绑定过程将使用同样的名字搜索查询串的键。

26.3.2　使用请求头作为绑定源

FromHeader特性允许将HTTP请求头作为数据绑定源。代码清单26-31在Home控制器中添加了一个简单的操作方法，用于从标准的HTTP请求头中接收参数。

代码清单26-31　在Controllers文件夹下的HomeController.cs文件中通过请求头进行绑定

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
```

```
    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index([FromQuery] int? id)
 {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public string Header([FromHeader]string accept)
 => $"Header: {accept}";
    }
}
```

Header方法定义了一个accept参数，当前请求中的Accept请求头的值将被接收并作为方法的结果返回。如果运行应用并访问/Home/Header，你将看到类似于下面的结果（虽然基于使用的浏览器不同，确切的执行结果可能也不同）：

```
Header: text/html,application/xhtml+xml,application/xml
;q=0.9,image/webp,*/*;q=0.8
```

并不是所有的HTTP请求头名称都可以依靠操作方法的参数名称易于选择，因为模型绑定系统不能使用HTTP请求头从C#命名约定进行转换。在这些情况下，必须使用Name属性配置FromHeader特性以指定请求头的名称，如代码清单26-32所示。

代码清单26-32　在Controllers文件夹下的HomeController.cs中指定请求头的名称

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index([FromQuery] int? id)
 {
            Person person;
```

```
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public string Header([FromHeader(Name = "Accept
-Language")] string accept)
            => $"Header: {accept}";
    }
}
```

不能使用Accept-Language作为C#参数名称，模型绑定器也不能自动将AcceptLanguage转换为Accept- Language以便匹配请求头。相反，这里使用Name属性来配置特性，进而匹配正确的请求头。如果启动应用并访问/Home/Header，你将看到类似下面的响应，这将基于不同的区域设置而有所变化。

```
Header: en-US,en;q=0.8
```

从请求头绑定复杂类型

尽管需求十分罕见，但仍然可以通过应用FromHeader特性到模型类的属性来使用请求头中的值绑定复杂类型。作为示例，在Models文件夹中添加名为HeaderModel.cs的类文件，定义代码清单26-33所示的类。

代码清单26-33　Models文件夹下的HeaderModel.cs文件的内容

```
using Microsoft.AspNetCore.Mvc;

namespace MvcModels.Models {

    public class HeaderModel {

        [FromHeader]
        public string Accept { get; set; }

        [FromHeader(Name = "Accept-Language")]
        public string AcceptLanguage { get; set; }

        [FromHeader(Name = "Accept-Encoding")]
        public string AcceptEncoding { get; set; }
    }
}
```

这个类定义了3个属性，其中的每个都使用

FromHeader特性进行了修饰。可在其中两个FromHeader特性中使用Name属性来指定不能表示为C#参数名的请求头名称。代码清单26-34更新了Home控制器的Header方法以接收HeaderModel对象。

代码清单26-34　在Controllers文件夹下的HomeController.cs文件中使用Header模型类

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index([FromQuery] int? id)
 {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
```

```
                return NotFound();
            }
        }

        public ViewResult Header(HeaderModel model) =>
View(model);
    }
}
```

为了完成示例，在Views/Home文件夹中添加名为Header.cshtml的视图文件，并添加代码清单26-35所示的标记内容。

代码清单26-35　Views/Home文件夹下的Header.cshtml文件的内容

```
@model HeaderModel
@{
    ViewBag.Title = "Headers";
    Layout = "_Layout";
}

<table class="table table-sm table-bordered table-strip
ed">
    <tr><th>Accept:</th><td>@Model.Accept</td></tr>
    <tr><th>Accept-Encoding:</th><td>@Model.AcceptEncod
ing</td></tr>
    <tr><th>Accept-Language:</th><td>@Model.AcceptLangu
age</td></tr>
</table>
```

模型绑定过程将检查复杂类型的属性以查找表26-3中的特性。如果运行应用并访问/Home/Header，你将会看到，这允许使用FromHeader 特性来定义从请求头绑定复杂类型的属性。产生的结果如图 26-10所示。

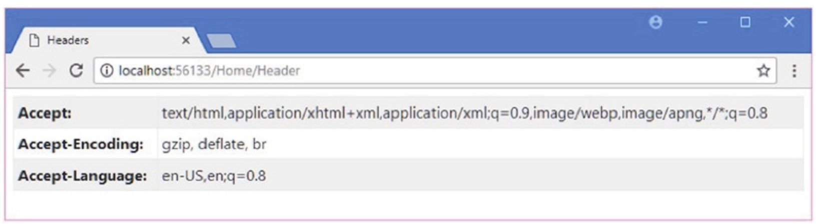


图26-10　从请求头中模型绑定复杂类型

26.3.3　使用请求体作为绑定源

不是所有的客户端数据都以表单形式发送，比如当JavaScript客户端发送JSON数据给API控制器时。FromBody特性指定请求体应当被解码并作为模型绑定源。代码清单26-36添加了名为Body的操作方法来演示具体如何工作。

代码清单26-36　在Controllers文件夹下的HomeController.cs文件中添加操作方法

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public IActionResult Index([FromQuery] int? id)
 {
            Person person;
            if (id.HasValue && (person = repository[id.
Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }

        public ViewResult Header(HeaderModel model) =>
View(model);

        public ViewResult Body() => View();

        [HttpPost]
        public Person Body([FromBody]Person model) => m
odel;
```

```
    }
}
```

这里使用FromBody特性修饰了Body方法的参数以接收POST请求，这意味着请求体的内容将被解码并用于模型绑定。参考第20章，MVC拥有可扩展的系统，但是默认仅处理JSON数据。

然后，编辑bower.json文件以添加jQuery包到应用中，如代码清单26-37所示。

代码清单26-37　在MvcModels文件夹下的bower.json文件中添加jQuery包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

为了提供操作方法需要的数据，添加一个名为

Body.cshtml的视图文件到Views/Home文件夹中，添加代码清单26-38所示的内容。

代码清单26-38　Views/Home文件夹下的Body.cshtml文件的内容

```
@{
    ViewBag.Title = "Address";
    Layout = "_Layout";
}

@section scripts {
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <script type="text/javascript">
        $(document).ready(function () {
            $("button").click(function (e) {
                $.ajax("/Home/Body", {
                    method: "post",
                    contentType: "application/json",
                    data: JSON.stringify({
                        firstName: "Bob",
                        lastName: "Smith"
                    }),
                    success: function (data) {
                        $("#firstName").text(data.first
Name);

                        $("#lastName").text(data.lastNa
me);
                    }
                });
```

```
            });
        });
    </script>
}

<table class="table table-sm table-bordered table-strip
ed">
    <tr><th>First Name:</th><td id="firstName"></td></t
r>
    <tr><th>Last Name:</th><td id="lastName"></td></tr>
</table>
<button class="btn btn-primary">Submit</button>
```

为了简化，Body视图包含一些内联的JavaScript代码，当button元素被单击时，使用jQuery发送包含JSON数据的HTTP POST请求到/Home/Body。服务器编码模型绑定创建的对象并编码为JSON后发送回客户端，运行应用并访问/Home/Body，然后单击Submit按钮，就可以看到效果，如图26-11所示。

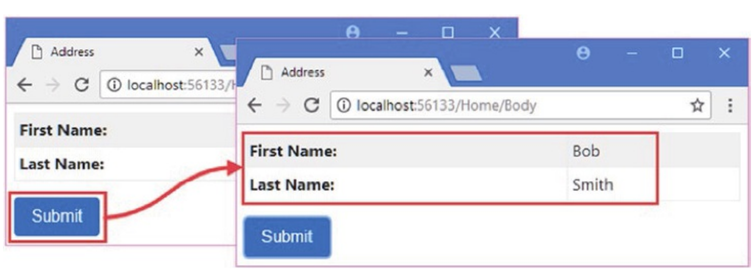


图26-11　将请求体用于模型绑定

提　示

不是所有的JavaScript客户端代码都需要使用FromBody特性。这个示例不得不避免使用jQuery便捷方法来发送Ajax POST请求，因为数据已编码为表单数据。相反，不得不使用其他方法来允许发送JSON数据。

FromBody特性只可以用于一个操作方法参数，如果这个特性被用于一个以上的操作方法参数，将会导致异常。如果需要为一个请求体创建多

个模型绑定对象，那么在操作方法中，你将不得不创建一个简单的拥有所有所需属性的数据传输类来创建对象。

26.4 小结

本章描述了模型绑定过程，模型绑定用来从HTTP请求中获取数据，以便为操作方法提供需要的参数。本章还解释了如何绑定简单类型和复杂类型，以及如何处理数组和集合，可通过应用特定到操作方法参数或模型类属性来控制模型绑定过程。下一章将描述模型验证功能。

第27章　模型验证

上一章展示了MVC如何通过模型绑定过程从HTTP请求中创建模型对象。本章介绍如何对用户提供的数据进行基本验证。实际情况是，用户经常输入无效或无用的数据，因此本章讨论模型验证。

模型验证是确认应用接收到的数据适合用来绑定到模型的过程，如果情况不是这样，则为用户提供有助于解释问题的有用信息。

要完成模型验证过程，第一步，检查接收到的数据是保持领域模型完整性的关键。拒绝从领域角度看没有意义的数据可以防止应用程序中出现奇怪或不希望的状态。第二步，帮助用户纠正问题也同样重要。如果没有用户需要的与应用程序进行交互的信息和反馈，用户就会变得沮丧和困惑。在面向

公众的应用中，这意味着用户会简单地放弃使用这个应用；在企业级应用中，这意味着用户的工作流程会阻滞。以上所有结果都是不可取的。但幸运的是，MVC为模型验证提供广泛的支持。表27-1给出了模型验证的背景。

表**27-1**　模型验证的背景

问题	答案
模型验证是什么？	模型验证是确保请求中提供的数据可以在应用中有效使用的过程
为什么模型验证是有用的？	用户不总是输入有效的数据，在应用中使用无效数据将导致无法预料或不期望的错误
如何使用模型验证？	控制器检查验证过程的结果，标签助手用来在用户显示的视图中包含验证反馈。验证在模型绑定过程中自动执行，通常在控制器类中或者通过使用验证特性来支持自定义验证
模型验证有何缺陷或局限性？	测试验证代码的有效性是非常重要的，确保它们能够阻止应用接收所有范围的值

模型验证有其他替代选择吗？	没有，模型验证它与ASP.NET Core MVC紧密集成

表27-2列出了本章要介绍的操作。

表**27-2**　本章要介绍的操作

操作	方法	代码清单
显式地验证模型	使用ModelState记录验证错误	代码清单27-9和代码清单27-10
汇总验证错误	为div元素应用asp-validation-summary属性	代码清单27-11
修改默认的模型绑定信息	在模型绑定信息提供器中重新定义消息函数	代码清单27-12
生成属性级别的验证错误信息	为span元素应用asp-validation-for属性	代码清单27-13
生成模型级别的验证错误信息	使用ModelState记录没有关联到特定属性的验证错误，为div元素的asp-validation-summary属性使用ModelOnly值	代码清单27-14和代码清单27-15
定义自验证模	为模型属性应用数据验证特性	代码清单27-16和代码清单27-

型		17
创建自定义验证特性	实现IModelValidator接口	代码清单27-18和代码清单27-19
执行客户端验证	使用jQuery验证和jQuery无痕验证包	代码清单27-20和代码清单27-21
执行远程验证	定义操作方法以执行验证，为模型属性应用Remote特性	代码清单27-22和代码清单27-23

27.1　准备示例项目

在本章中，使用ASP.NET Core Web Application（.NET Core）模板创建名为ModelValidation的Empty项目。代码清单27-1展示了Startup类，这里添加了MVC框架并启用了开发中用到的中间件。

代码清单27-1　ModelValidation文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ModelValidation {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

27.1.1 创建模型

首先创建Models文件夹，添加一个名为

Appointment.cs的类文件，并用它定义代码清单27-2所示的类。

代码清单27-2　Models文件夹下的Appointment.cs文件的内容

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
    public class Appointment {

        public string ClientName { get; set; }

        [UIHint("Date")]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

Appointment模型类定义了3个属性，还使用了UIHint特性来指示Date属性应该展示为没有时间部分的日期。

27.1.2　创建控制器

然后创建Controllers文件夹，添加一个名为HomeController.cs的类文件，并用它定义代码清单27-3所示的控制器，用以处理Appointment模型类。

代码清单27-3　Controllers文件夹下的HomeController.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {

    public class HomeController : Controller {
        public IActionResult Index() =>
            View("MakeBooking", new Appointment { Date
= DateTime.Now });

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
 =>
            View("Completed", appt);
    }
}
```

Index方法使用新的Appointment对象作为视图模型来渲染MakeBooking视图。在本章中，

MakeBooking方法更有意义，因为模型验证将在这个操作方法中处理。

注　意

这个应用十分简单，既没有定义存储库，也没有添加任何代码来存储模型绑定过程中产生的Appointment对象。这就是说，需要铭记在心的是，验证模型的主要目的是防止坏的或无意义的数据被放置到存储库中并导致问题（无论是在试图存储数据时还是以后试图处理数据时）。

27.1.3　创建布局和视图

在本章中，对于某些示例使用需要简单的布局。创建Views/Shared文件夹，在其中添加名为_Layout.cshtml的视图文件，内容如代码清单27-4所示。

代码清单27-4　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width"
/>
    <title>Model Validation</title>
    <link asp-href-include="/lib/bootstrap/dist/css/boo
tstrap.min.css" rel="stylesheet" />
    @RenderSection("scripts", false)
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

下面为操作方法提供视图。创建Views/Home文件夹，并添加名为MakeBooking.cshtml的视图文

件，内容如代码清单27-5所示。

代码清单27-5　Views/Home文件夹下的MakeBooking.cshtml文件的内容

```
@model Appointment

@{ Layout = "_Layout"; }

<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
```

```
Booking</button>
</form>
```

当Index.cshtml文件中的表单被回发到应用程序时，MakeBooking方法使用Views/Home文件夹中的Completed视图显示用户创建的预约详情，如代码清单27-6所示。

代码清单27-6　Views/Home文件夹下的Completed.cshtml文件的内容

```
@model Appointment
@{ Layout = "_Layout"; }

<div class="bg-success m-1 p-1 text-white"><h2>Your App
ointment</h2></div>

<table class="table table-bordered">
    <tr>
        <th>Your name is:</th>
        <td>@Model.ClientName</td>
    </tr>
    <tr>
        <th>Your appontment date is:</th>
        <td>@Model.Date.ToString("d")</td>
    </tr>
</table>
<a class="btn btn-success" asp-action="Index">Make Anot
her Appointment</a>
```

Completed视图基于Bootstrap CSS包来修饰HTML元素。下面为项目添加Bootstrap包，使用Bower Configuration File模板创建bower.json文件，在dependencies部分添加Bootstrap包，如代码清单27-7所示，这里还添加了jQuery包到项目中，本章后面会用到jQuery。

代码清单27-7　添加Bootstrap包到bower.json文件中

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

最后的准备工作是在Views文件夹中创建_ViewImports.cshtml文件，在其中设置内置的标签助手用于Razor视图，并导入模型命名空间，如代码清单27-8所示。

代码清单27-8　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using ModelValidation.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

如你所知，本章的示例围绕着如何创建预约。可以通过启动应用并且访问默认URL地址来看看具体是如何工作的。在表单中输入预约详情，单击Make Booking按钮将数据发送回服务器，执行模型绑定过程以创建Appointment对象，预约详情将使用Completed视图渲染出来，如图27-1所示。

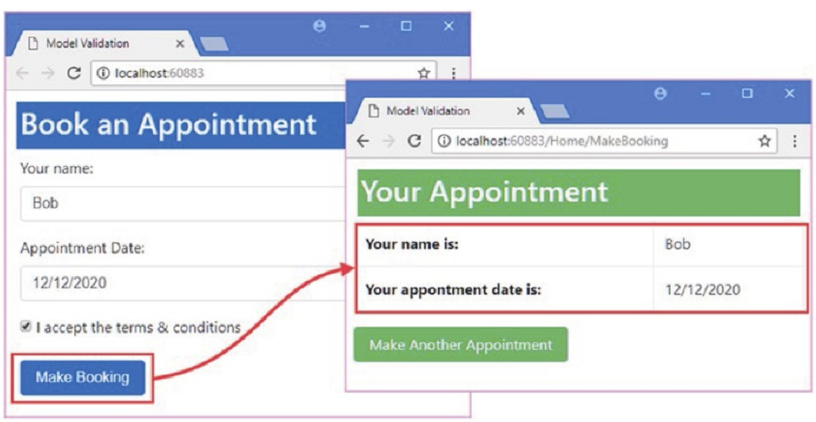


图27-1　使用示例应用

27.2　理解模型验证的需求

模型验证是确保应用程序从客户端接收的数据满足要求的过程。没有验证，应用程序将会试图处理接收的任何数据，这将导致异常或意外的行为，还将面临随着存储中填充不良数据，不完整或恶意的数据逐渐出现的长期问题。

目前，应用会接收用户提交的任何数据。为保护应用和领域模型的完整性，在知道用户提交可接收的预约对象之前，需要做以下3件事：

- 提供名字。
- 提供未来的日期。
- 选中复选框以接受条款。

下面将演示如何使用模型验证，通过检查应用接收的数据，以及在应用不能使用提交的数据时为用户提供反馈来强制满足这些要求。

27.3　显式地验证模型

多数直接验证模型的途径定义在操作方法中，在代码清单27-9所示的MakeBooking方法中，已加入对Appointment类的每个属性的显式检查。

代码清单27-9　在Controllers文件夹下的HomeController.cs文件中显式地验证模型

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() =>
            View("MakeBooking", new Appointment { Date
= DateTime.Now });

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
 {
            if (string.IsNullOrEmpty(appt.ClientName)) {

                ModelState.AddModelError(nameof(appt.Cl
ientName),
```

```
                    "Please enter your name");
            }

            if (ModelState.GetValidationState("Date")
                    == ModelValidationState.Valid && Da
teTime.Now > appt.Date) {
                ModelState.AddModelError(nameof(appt.Da
te),
                    "Please enter a date in the future"
);
            }
            if (!appt.TermsAccepted) {
                ModelState.AddModelError(nameof(appt.Te
rmsAccepted),
                    "You must accept the terms");
            }

            if (ModelState.IsValid) {
                return View("Completed", appt);
            } else {
                return View();
            }
        }
    }
}
```

检查模型绑定器已经赋予参数对象的属性值，并使用从控制器基类继承的ModelState属性返回的ModelStateDictionary类型对象来注册任何错误。

顾名思义，ModelStateDictionary类是用来跟踪

模型对象状态详情的字典，主要用于验证错误，表27-3描述了ModelStateDictionary类的重要成员。

表**27-3　ModelStateDictionary**类的重要成员

成员	描述
AddModelError(property,message)	这个方法用于对特定的属性记录模型验证错误
GetValidationState(property)	这个方法用来检测特定的属性是否存在模型验证错误，表示为ModelValidationState枚举中的值
IsValid	如果所有的模型属性都是有效的，那么这个属性返回true；否则，返回false

作为使用ModelStateDictionary的示例，考虑ClientName属性是如何被验证的：

```
...
if (string.IsNullOrEmpty(appt.ClientName)) {
    ModelState.AddModelError(nameof(appt.ClientName), "
Please enter your name");
}
...
```

以上示例中，验证的目标之一是确认用户为这个属性提供了有效值，所以可使用string.IsNullOrEmpty静态方法来测试模型绑定已经从请求中提取的属性值。如果ClientName是null或空串，就会得出验证目的没有达成的结论，然后使用ModelState.AddModelError方法注册验证错误信息，指定属性的名字（ClientName）以及将会用来显示给用户以说明问题原因的信息（Please enter your name）。

模型绑定系统还使用ModelStateDictionary记录查找和分配值给模型属性的任何问题。GetValidationState方法用于查看模型属性是否存在任何错误，无论是从模型绑定过程还是由于在操作方法中进行显式验证期间调用了AddModelError方法。GetValidationState方法会返回ModelValidationState枚举中的一个值，如表27-4所

示。

表**27-4　ModelValidationState**枚举中的值

值	描述
Unvalidated	这个值意味着没有在模型属性上执行验证，通常是因为在请求中没有值关联到属性名称
Valid	这个值意味着关联到属性的请求值是有效的
Invalid	这个值意味着关联到属性的请求值是无效的，并且不能使用
Skipped	这个值意味着模型属性还没有处理，通常意味着出现了太多的验证错误，以至于无法继续执行验证检查

对于Date属性，检查模型绑定过程是否报告了关于将浏览器发送的值解析为DateTime对象的错误，如下所示：

```
...
if (ModelState.GetValidationState("Date") == ModelValid
ationState.Valid
        && DateTime.Now > appt.Date) {
    ModelState.AddModelError(nameof(appt.Date), "Please
 enter a date in the future");
```

```
}
...
```

对于Date属性，验证目标是确认用户提供了正确的未来日期。使用GetValidationState方法检查Model- ValidationState.Valid的值以判定模型绑定过程能够解析请求的值到DateTime对象。如果存在有效的日期，再确认是正确的未来日期，如果不是，就使用AddModelError方法记录存在验证问题。

在验证模型对象的所有属性之后，检查ModelState.IsValid属性以查看是否存在错误。如果在检查过程中ModelState.AddModelError方法被调用了，或者模型绑定器在创建Appointment对象时出现问题，这个方法将返回true。

```
...
if (ModelState.IsValid) {
    return View("Completed", appt);
} else {
    return View();
}
```

```
...
```

如果IsValid属性返回true，那么Appointment对象是有效的，在这种情况下，操作方法渲染Completed视图。如果IsValid属性返回false，就表示出现验证问题，可通过调用View方法来渲染默认视图。

27.3.1　为用户显示验证错误消息

通过调用View方法来处理验证错误消息看起来很奇怪，MVC提供给视图的上下文数据中包含模型验证错误的详细信息，它们被自动检测并由用于转换input元素的标签助手使用。

为了查看具体是如何工作的，启动应用并在不填充任何表单内容的情况下单击Make Booking按钮。看起来没有任何浏览器窗口中的可视内容发生变化。但是，如果查看MVC为这个POST请求返回

的HTML内容，你将看到表单元素的class属性发生了变化。以下是ClientName元素在提交表单之前的内容：

```
<input class="form-control" type="text" id="ClientName"
 name="ClientName" value="">
```

以下是提交空的表单之后的input元素：

```
<input class="form-control input-validation-error" type
="text" id="ClientName"
    name="ClientName" value="">
```

标签助手为验证失败的元素添加了input-validation-error样式类，然后设置为向用户突出显示问题。这可以通过在样式表中定制CSS样式来实现，但是如果希望使用内置的样式表，比如Bootstrap提供的样式表，那么还有一点额外的工作要做。添加到form元素的样式类名不能修改，这意味着某些JavaScript代码需要在MVC使用的名称和Bootstrap提供的CSS错误类名之间进行映射。

提　示

使用这样的JavaScript代码可能很尴尬，即使使用Bootstrap这样的样式库，也可以使用自定义样式。然而，Bootstrap中用于验证类的颜色可以使用主题或者通过定制包以及定义自己的样式来覆盖。这意味着你将不得不确认对于主题的任何修改都与自定义样式的相应更改相匹配。理想情况下，微软将使未来发行版本的ASP.NET Core MVC中验证类的名称可配置，但在此之前，使用JavaScript来应用Bootstrap样式是比创建自定义样式更加强大的方法。

代码清单27-10添加了jQuery代码到MakeBooking视图以查找拥有inputvalidation-error样式类的元素，定位最临近的被赋予form-group样式类的父元素，然后为这个元素添加has-error样式类（Bootstrap用它为form元素设置出错颜色）。

代码清单27-10　在Views/Home文件夹下的MakeBooking.cshtml文件中赋予元素验证类

```
@model Appointment

@{ Layout = "_Layout"; }

@section scripts {
    <script src="/lib/jquery/dist/jquery.min.js"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            $("input.input-validation-error")
                .closest(".form-group").addClass("has-danger");
        });
    </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>
```

```
<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
Booking</button>
</form>
```

在浏览器完成解析HTML文档中的所有元素之后，jQuery代码被执行，效果是突出显示被赋予input- validation-error样式类的input元素。可以通过运行应用并在没有填写任何文本框的情况下提交表单来查看效果，处理的结果如图27-2所示。

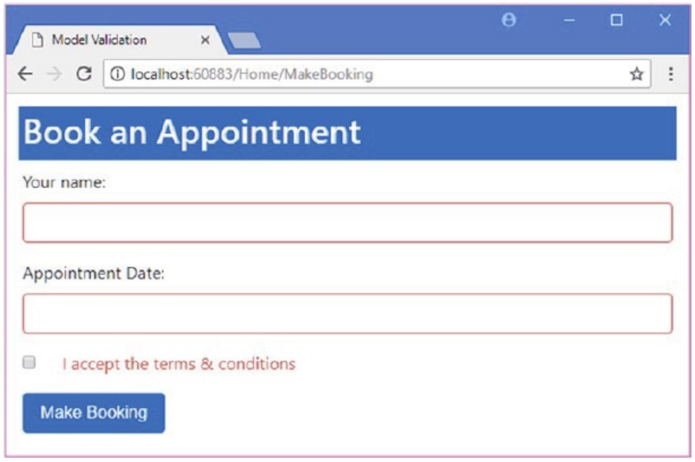


图27-2　高亮显示验证错误信息

可在没有输入任何数据的情况下提交表单，对于3个文本框，错误消息都将突出显示。除非提交的表单可以被模型绑定器解析，并且通过了MakeBooking方法的显式验证，否则用户将看不到Completed视图。在此之前，提交表单将会导致使用当前验证错误消息的MakeBooking视图被渲染。

27.3.2　显示验证消息

一些标签助手应用到input元素的CSS样式类标志着表单字段存在问题，但是它们没有告诉用户是

什么问题。要为用户提供更加详尽的消息，可使用其他标签助手添加问题的摘要到视图中，如代码清单27-11所示。

代码清单27-11　在Views/Home文件夹下的MakeBooking.cshtml文件中显示验证摘要

```
@model Appointment

@{ Layout = "_Layout"; }

@section scripts {
<script src="/lib/jquery/dist/jquery.min.js"></script>
<script type="text/javascript">
    $(document).ready(function () {
        $("input.input-validation-error")
            .closest(".form-group").addClass("has-dange
r");
    });
</script>
}
<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div asp-validation-summary="All" class="text-dange
r"></div>
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
```

```
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
Booking</button>
</form>
```

ValidationSummaryTagHelper类监测div元素的asp-validation-summary属性，通过添加描述由操作方法检测的任何验证错误消息进行响应。asp-validation-summary 属性的值来自ValidationSummary 枚举，如表27-5所示。

表**27-5　ValidationSummary**枚举中的值

值	描述

All	用来显示所有被记录的验证错误
ModelOnly	用来显示仅限于整个模型的验证错误，不包括被记录的属于独立属性的验证错误
None	用来禁用标签助手以便不转换HTML元素

如果运行应用，并在没有做任何修改的情况下提交表单，你将会看到标签助手生成的摘要信息。本示例中的文本颜色由text-dangerBootstrap类定义，该类确保文本匹配文本框中突出显示的颜色，如图27-3所示。

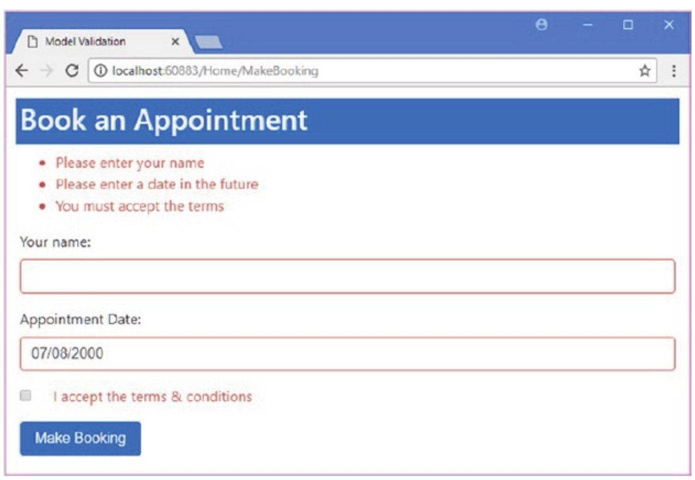


图27-3　为用户显示验证摘要时文本框中突出显示的颜色

如果查看浏览器接收到的HTML，你将会看到验证信息被作为列表发送，像下面这样：

```
<div class="text-danger validation-summary-errors" data
-valmsg-summary="true">
  <ul>
    <li>Please enter your name</li>
    <li>Please enter a date in the future</li>
    <li>You must accept the terms</li>
  </ul>
</div>
```

配置默认验证错误消息

当第26章描述的模型绑定过程尝试提供调用操作方法所需的数据时，会执行自己的验证。要了解具体是如何工作的，可启动应用，清除Appointment Date文本框中的内容，然后提交表单。你将看到显示的错误消息之一已经变化，并且与操作方法中传递给AddModelError方法的任何字符串都不匹配。

```
The value '' is invalid
```

当模型绑定过程找不到属性的值或者虽然找到值但是不能解析时，该消息将被添加到ModelState-Dictionary 中。在这个示例中，出现错误是因为表单中提供的空字符串不能解析为Appointment对象的Date属性要求的DateTime对象。

模型绑定器有一组用于验证错误的预定义信息。通过将函数赋予由DefaultModelBindingMessageProvider类定义的方法，可以将它们替换为自定义的消息，如表27-6所示。

表**27-6　DefaultModelBindingMessageProvider**类定义的方法

方法	描述
SetValueMustNotBeNullAccessor	当非空的模型属性值为空时，生成验证错误消息
SetMissingBindRequiredValueAccessor	当请求中没有包含必需属性的值时，生成验证错误消息

SetMissingKeyOrValueAccessor	当字典模型对象要求的数据包含null键或值时，生成验证错误消息
SetAttemptedValueIsInvalidAccessor	当尝试提供给模型绑定系统的值无效时，生成验证错误消息
SetUnknownValueIsInvalidAccessor	当模型绑定系统不能将数据值转换为要求的C#类型时，生成验证错误消息
SetValueMustBeANumberAccessor	当数据值不能被解析为C#数值类型时，生成验证错误消息
SetValueIsInvalidAccessor	生成备用的验证错误消息，用作最后的手段

表27-6中描述的每个方法接收一个函数，它将被调用以获取显示给用户的验证消息。这些方法用于在Startup类中配置应用，代码清单27-12替换了当值为null时的默认消息。

代码清单27-12　在ModelValidation文件夹下的Startup.cs文件中替换绑定的消息

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
namespace ModelValidation {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc().AddMvcOptions(opts =>
                opts.ModelBindingMessageProvider
                    .SetValueMustNotBeNullAccessor(valu
e => "Please enter a value")
            );
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

可指定用来接收用户提供的值的函数，尽管这对于null值并不是特别有帮助。为了改变模型绑定的错误消息，可重新启动应用并在清空Appointment Date文本框之后提交表单，如图27-4所示。

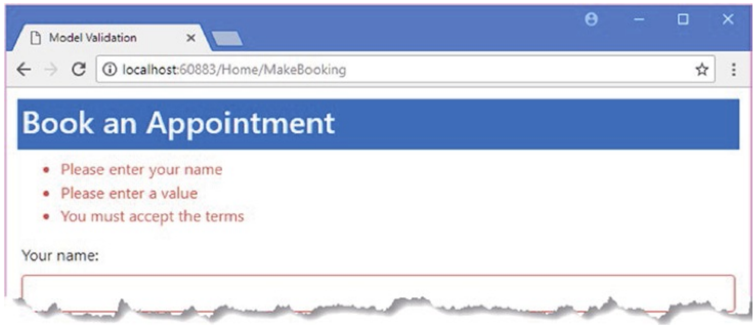


图27-4　改变模型绑定的错误消息

27.3.3　显示属性级验证消息

尽管自定义的错误消息比默认的更有意义，但它仍然没有帮助，因为它没有清楚地为用户标识问题。对于这种错误，在包含错误数据的HTML元素旁显示错误消息更有帮助，这可以使用ValidationMessageTag标签助手来完成，该标签助手将查找拥有asp-validation-for属性的span元素，该元素用于指定应显示错误消息的模型属性。

代码清单27-13为表单中的每个input元素添加了属性级的验证消息元素。这里还删除了scripts部

分，因为独立的验证消息将使验证错误的元素足够醒目。

代码清单27-13　在Views/Home文件夹下的MakeBooking.cshtml文件中添加属性级验证消息

```
@model Appointment

@{ Layout = "_Layout"; }

@section scripts {
<script src="/lib/jquery/dist/jquery.min.js"></script>
<script type="text/javascript">
    $(document).ready(function () {
        $("input.input-validation-error")
            .closest(".form-group").addClass("has-dange
r");
    });
</script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div asp-validation-summary="All" class="text-dange
r"></div>
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
        <div><span asp-validation-for="ClientName" clas
```

```
s="text-danger"></span></div>
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <div><span asp-validation-for="Date" class="tex
t-danger"></span></div>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <span asp-validation-for="TermsAccepted" class="tex
t-danger"></span>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
Booking</button>
</form>
```

由于span元素作为行内元素显示，因此必须注意提供的验证消息与哪个元素相关。通过运行应用程序并在不输入任何数据的情况下提交表单，可以看到新的验证消息的影响，如图27-5所示。

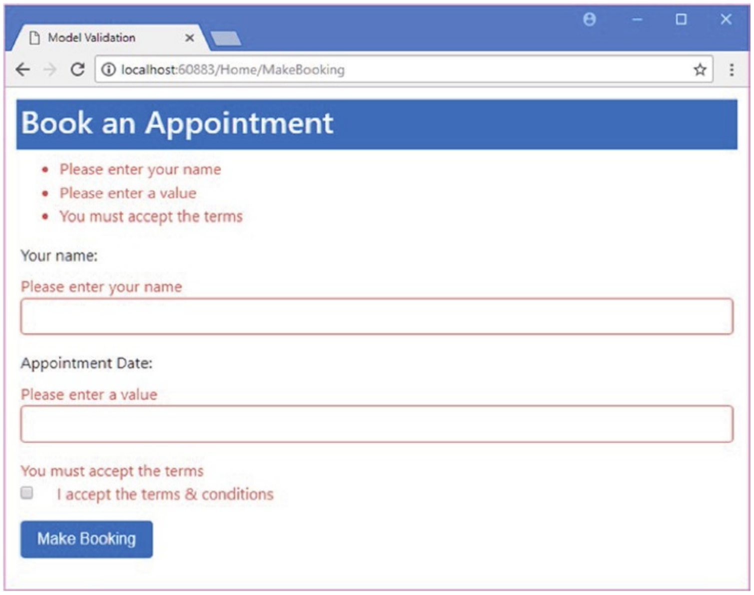


图27-5　使用属性级验证消息

27.3.4　显示模型级验证消息

验证摘要信息似乎是多余的，因为它们仅仅重复了属性级验证信息。属性级验证信息由于出现在需要解决问题的表单元素旁边，通常对用户更有帮助。但是验证摘要信息有显示用于整个模型的信息的能力，这意味着可以报告由组合的单个属性引起的错误，例如，在给定的日期仅仅对特定名称的组合才有效的情况下。

代码清单27-14添加了验证检查，用来阻止名为Joe的客户于星期一的预约。

代码清单27-14　在Controllers文件夹下的HomeController.cs文件中执行模型级验证

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() =>
            View("MakeBooking", new Appointment() { Dat
e = DateTime.Now });
        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
 {
            if (string.IsNullOrEmpty(appt.ClientName))
{
                ModelState.AddModelError(nameof(appt.Cl
ientName),
                    "Please enter your name");
            }

            if (ModelState.GetValidationState("Date")
                    == ModelValidationState.Valid && Da
teTime.Now > appt.Date) {
```

```
                ModelState.AddModelError(nameof(appt.Da
te),
                    "Please enter a date in the future"
);
            }

            if (!appt.TermsAccepted) {
                ModelState.AddModelError(nameof(appt.Te
rmsAccepted),
                    "You must accept the terms");
            }

            if (ModelState.GetValidationState(nameof(ap
pt.Date))
                    == ModelValidationState.Valid
                && ModelState.GetValidationState(nameof
(appt.ClientName))
                    == ModelValidationState.Valid
                && appt.ClientName.Equals("Joe", String
Comparison.OrdinalIgnoreCase)
                && appt.Date.DayOfWeek == DayOfWeek.Mon
day) {
                    ModelState.AddModelError("",
                        "Joe cannot book appointments o
n Mondays");
            }

            if (ModelState.IsValid) {
                return View("Completed", appt);
            } else {
                return View();
            }
        }
    }
}
```

这段代码看起来比实际上要复杂，这是数据验证的特点。通过检查模型状态，确认已经收到有效的ClientName和Date值，然后检查特定的日期是否为星期一，以及ClientName属性的值是否为Joe。如果Joe试图于星期一预约，那么使用空字符串（""）作为第一个参数来调用AddModelError方法，这表示错误将应用于整个模型而不是单个属性。

代码清单27-15通过将asp-validation-summary属性的值改为ModelOnly，排除了属性级别的错误消息，这意味着摘要将仅仅显示应用于整个模型的错误消息。

代码清单27-15　在Views/Home文件夹下的MakeBooking.cshtml文件中显示模型级验证错误

```
@model Appointment

@{ Layout = "_Layout"; }

@section scripts {
```

```
<script src="/lib/jquery/dist/jquery.min.js"></script>
<script type="text/javascript">
    $(document).ready(function () {
        $("input.input-validation-error")
            .closest(".form-group").addClass("has-dange
r");
    });
</script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div asp-validation-summary="ModelOnly" class="text
-danger"></div>
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
        <div><span asp-validation-for="ClientName" clas
s="text-danger"></span></div>
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <div><span asp-validation-for="Date" class="tex
t-danger"></span></div>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <span asp-validation-for="TermsAccepted" class="tex
t-danger"></span>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
```

```
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
Booking</button>
</form>
```

通过运行应用可以查看效果，输入Joe到Your name文本框中，选择预约日期为01/18/2027，提交表单，你将看到图27-6所示的响应。

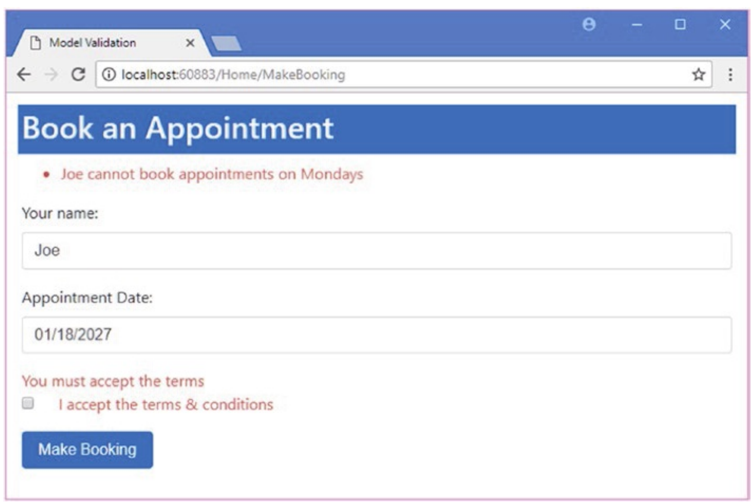


图27-6　响应

27.4　使用元数据指定验证规则

将验证逻辑放入操作方法的问题是会在每个接

收数据的操作方法中导致重复。为了减少重复，验证过程支持使用特性直接在模型类中表达验证规则，确保无论哪个操作方法用来处理请求，都使用相同的一组验证规则，

代码清单27-16已在Appointment类中应用验证特性以实施属性级验证规则。

代码清单27-16　在Models文件夹下的Appointment.cs文件中应用验证特性

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {

    public class Appointment {

        [Required]
        [Display(Name = "name")]
        public string ClientName { get; set; }

        [UIHint("Date")]
        [Required(ErrorMessage = "Please enter a date")]

        public DateTime Date { get; set; }
        [Range(typeof(bool), "true", "true",
```

```
        ErrorMessage = "You must accept the terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

以上代码使用了两个验证特性——Required和Range。Required特性指定了如果用户没有为属性提供值，就是验证错误。Range特性指定了可接收的子集。表27-7展示了MVC应用中内置的验证特性。

表**27-7**　内置的验证特性

特性	示例	描述
Compare	[Compare("OtherProperty")]	这个特性确保属性拥有相同的值，当要求用户提供两次相同的信息时很有用，比如电子邮件地址或口令
Range	[Range(10,20)]	这个特性确保数值（或者任何实现了IComparable的属性类型）不超出指定的最小值和最大值。为了仅指定单侧边界，可使用MinValue或MaxValue常量，例如[Range(int.MinValue,50)]
RegularExpression	[RegularExpression	这个特性确保字符串值与指定的正则表达式匹配。注意，模式必须匹配整个用户提供的值，而不仅仅是其中的子串。默认情

	("pattern")]	况下，匹配时区分大小写，但可以通过应用（?i）修饰符（如[RegularExpression("(?i)mypattern")]）来不区分大小写
Required	[Required]	这个特性确保值不为空或是仅仅包含空格的字符串。如果要将空格视为有效，请使用[Required（AllowEmptyStrings = true）]
StringLength	[StringLength(10)]	这个特性确保字符串的值不长于指定的最大长度，还可以指定最小长度，如[StringLength(10,MinimumLength=2)]

所有的验证特性都支持通过为ErrorMessage属性设置值来定制错误消息，类似于下面这样：

```
...
[UIHint("Date")]
[Required(ErrorMessage = "Please enter a date")]
public DateTime Date { get; set; }
...
```

如果这里没有定制错误消息，那么默认的信息将被应用，但是它们倾向于显示对用户没有意义的模型类的详细信息，除非还使用Display特性，并组合应用于ClientName属性。

```
...
[Required]
[Display(Name = "name")]
public string ClientName { get; set; }
...
```

通过Required特性生成的默认消息反映了使用Display特性指定的名称，因此不会将属性的名称显示给用户。

需要注意使验证保持一致，例如，考虑应用于TermAccepted属性的特性：

```
...
[Range(typeof(bool), "true", "true", ErrorMessage="You must accept the terms")]
public bool TermsAccepted { get; set; }
...
```

虽然十分希望用户选中复选框并接受条款，但不能使用Required特性，因为如果用户没有选中复选框，浏览器将为这个属性发送false值。为了绕过这个问题，可使用Range特性的一项功能：提供

Type，并将上限和下限指定为字符串值。通过设置两个界限都为true，为基于复选框的bool属性创建Required特性的等价物。这可能需要做一些实验，以确保浏览器发送的验证特性和数据一起工作。

在模型类上使用验证特性意味着控制器中的操作方法可以简化，如代码清单27-17所示。

代码清单27-17　在Controllers文件夹下的HomeController.cs文件中移除属性级验证

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() =>
            View("MakeBooking", new Appointment() { Dat
e = DateTime.Now });

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
 {
```

```
            if (ModelState.GetValidationState(nameof(ap
pt.Date))
                    == ModelValidationState.Valid
                && ModelState.GetValidationState(nameof
(appt.ClientName))
                    == ModelValidationState.Valid
                && appt.ClientName.Equals("Joe", String
Comparison.OrdinalIgnoreCase)
                && appt.Date.DayOfWeek == DayOfWeek.Mon
day) {
                ModelState.AddModelError("",
                    "Joe cannot book appointments on Mo
ndays");
            }

            if (ModelState.IsValid) {
                return View("Completed", appt);
            } else {
                return View();
            }
        }
    }
}
```

验证特性在调用操作方法之前被应用，这意味着可以仍然基于模型来确定在执行模型验证时单个属性是否有效。下面在操作方法中查看验证特性，启动应用，然后不输入任何数据就提交表单，如图27-7所示。

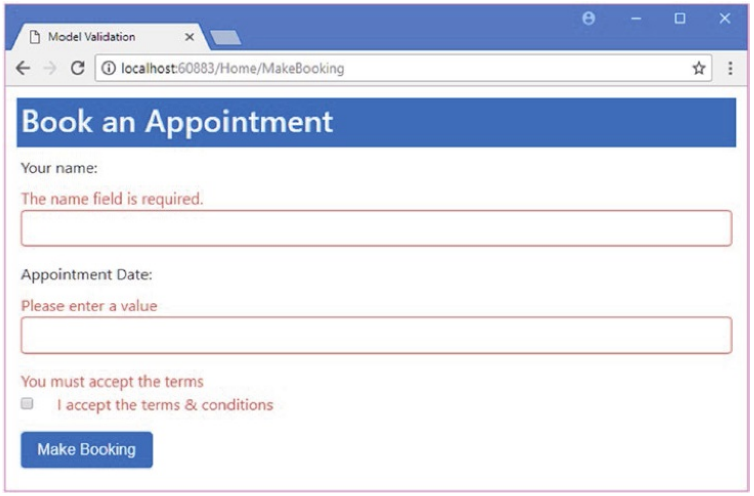


图27-7　使用验证特性

创建自定义的属性验证特性

验证过程可以通过创建实现了IModelValidator接口的特性来扩展。为了展示，创建Infrastructure文件夹，然后添加一个名为MustBeTrueAttribute.cs的类文件，定义代码清单27-18所示的类。

代码清单27-18　Infrastructure文件夹下的MustBeTrueAttribute.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
```

```
namespace ModelValidation.Infrastructure {
    public class MustBeTrueAttribute : Attribute, IMode
lValidator {

        public bool IsRequired => true;

        public string ErrorMessage { get; set; } = "Thi
s value must be true";

        public IEnumerable<ModelValidationResult> Valid
ate(
                ModelValidationContext context) {
            bool? value = context.Model as bool?;
            if (!value.HasValue || value.Value == false
) {
                return new List<ModelValidationResult>
{
                    new ModelValidationResult("", Error
Message)
                };
            } else {
                return Enumerable.Empty<ModelValidation
Result>();
            }
        }
    }
}
```

IModelValidator接口定义了名为IsRequired的属性，用于指示是否需要验证该类（这里有一点误导，因为这个属性返回的值仅仅被用于排序验证特

性，以便需要的先执行）。Validate方法用来执行验证并通过ModelValidationContext类的实例接收信息，该类定义的主要属性如表27-8所示。

表**27-8　ModelValidationContext**类定义的属性

属性	描述
Model	返回被验证的属性值，在示例中是TermsAccepted属性的值
Container	返回包含属性的对象，在示例中是Appointment对象
ActionContext	返回ActionContext对象，该对象提供上下文数据，并描述将处理请求的操作方法
ModelMetadata	返回ModelMetadata对象，该对象描述用于进行详细验证的模型类

Validate方法返回一个ModelValidationResult对象序列，其中每一个对象描述了一个验证错误。在示例特性中，如果模型值不是真的，就创建ModelValidationResult。ModelValidationResult构造函数的第一个参数是属性的名称，该属性与错误是

相关联的。当验证每个错误的时候，指定该属性是空字符串。错误消息的第二个参数是显示给用户的错误消息。在代码清单27-19中，用自定义特性替换了Range特性。

代码清单27-19　在Models文件夹中应用Appointment.cs文件中的自定义验证特性

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {

    public class Appointment {

        [Required]
        [Display(Name = "name")]
        public string ClientName { get; set; }

        [UIHint("Date")]
        [Required(ErrorMessage = "Please enter a date")
]
        public DateTime Date { get; set; }

        [MustBeTrue(ErrorMessage = "You must accept the
 terms")]
        public bool TermsAccepted { get; set; }
    }
```

```
}
```

使用自定义的验证特性的结果与使用Range特性的结果相同，但是在阅读代码时，自定义特性更加容易理解。

27.5　执行客户端验证

到目前为止，已展示的验证技术都是服务器端验证方面的，这意味着用户提交数据到服务器，然后服务器验证数据并返回验证结果（不管是成功处理数据还是需要更正的错误列表）。

在Web应用中，用户通常需要立即得到验证回馈，而无须向服务器提交任何内容，这称为客户端验证，并使用JavaScript来实现。用户输入的数据在发送到服务器之前被验证，为用户提供即时反馈并有机会更正任何错误。

MVC支持无痕的客户端验证。无痕意味着使用视图生成的、添加到HTML元素的属性来表示验证规则。这些属性由作为MVC一部分的JavaScript库进行解释，反过来，配置执行实际验证工作的jQuery验证库。后面将展示内置的验证支持如何工作，并展示如何扩展它的功能以支持自定义的客户端验证。

提　示

客户端验证的重点是验证单个属性。事实上，使用MVC的内置支持很难设置模型级客户端验证。为此，大多数MVC应用程序使用客户端验证来处理属性级别的问题，并依赖服务器端验证处理整体模

型。

第一步是使用Bower将新的JavaScript包加入应用，如代码清单27-20所示。

代码清单27-20　在ModelValidation文件夹下的bower.json文件中添加包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

使用客户端验证意味着将3个JavaScript文件——jQuery库、jQuery验证库和微软无痕验证库添加到视图中，所有这些如代码清单27-21所示。

提　示

Bower工具不会始终正确地安装验证包。如果你发现wwwroot/lib文件夹不包含需要的文件，那么可以删除wwwroot/lib及其内容。打开新的PowerShell窗口，找到项目文件夹，运行bower cache clean和bower install，下载验证包的新副本。

代码清单27-21　在Views/Home文件夹下的MakeBooking.cshtml文件中添加验证脚本元素

```
@model Appointment

@{ Layout = "_Layout"; }

@section scripts {
```

```
    <script src="/lib/jquery/dist/jquery.min.js"></scri
pt>
    <script src="/lib/jquery-validation/dist/jquery.val
idate.min.js"></script>
    <script
        src="/lib/jquery-validation-unobtrusive/jquery.
validate.unobtrusive.min.js">
    </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an
Appointment</h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="
post">
    <div asp-validation-summary="ModelOnly" class="text
-danger"></div>
    <div class="form-group">
        <label asp-for="ClientName">Your name:</label>
        <div><span asp-validation-for="ClientName" clas
s="text-danger"></span></div>
        <input asp-for="ClientName" class="form-control
" />
    </div>
    <div class="form-group">
        <label asp-for="Date">Appointment Date:</label>
        <div><span asp-validation-for="Date" class="tex
t-danger"></span></div>
        <input asp-for="Date" type="text" asp-format="{
0:d}" class="form-control" />
    </div>
    <span asp-validation-for="TermsAccepted" class="tex
t-danger"></span>
    <div class="radio form-group">
        <input asp-for="TermsAccepted" />
        <label asp-for="TermsAccepted" class="form-chec
```

```
k-label">
            I accept the terms & conditions
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Make
Booking</button>
</form>
```

这些文件必须按照顺序添加。当使用标签助手转换input元素的时候，它们会检查应用与模型类的验证特性并向输出元素添加属性。如果运行应用并检查发送到浏览器的HTML，你将会看到类似于下面这样的元素：

```
<input class="form-control" type="text" data-val="true"
    data-val-required="The name field is required." id=
"ClientName"
    name="ClientName" value="" />
```

JavaScript代码使用data-val属性来查找元素，并在用户提交表单时在浏览器中执行本地验证，而不向服务器发送HTTP请求。在使用F12键时，可以通过运行应用和提交表单来查看效果。注意，即使

把HTTP请求发送到服务器，也会显示验证错误消息。

避免与浏览器验证发生冲突

当前的一些HTML5浏览器支持基于input元素属性的简单客户端验证。例如，如果为input元素应用了required属性，那么当用户试图在不提供值的情况下提交表单时，会导致浏览器显示验证错误。

如果从模型中生成表单元素，那么浏览器验证将不会有任何问题，因为MVC将生成和使用data-属性来表示验证规则（例如，为必须具有值的input元素使用data-val-required属性，浏览器将不会识别该元素）。

但是，如果无法完全控制应用程序中的标记，

那么你可能会遇到问题。当你处理以其他途径生成的内容时，通常会发生这种错误。结果是jQuery验证和浏览器验证同时作用于表单，对用户造成困扰。为避免此类问题，可以将novalidate属性添加到form元素中。

MVC客户端验证的一个很好特性是用来指定验证规则的属性可同时应用于客户端和服务器端，这意味着来自不支持JavaScript的浏览器的数据将受同样的验证，而不需要做任何额外的工作。但是，这也意味着客户端验证不支持自定义的验证特性，因为JavaScript无法在客户端实现自定义验证逻辑。换句话说，如果要使用客户端验证，则需要遵守表27-7中描述的内置特性。

比较MVC客户端验证与jQuery验证

MVC客户端验证建立在jQuery验证库之上。如果愿意，你可以直接使用验证库而忽略MVC特性。验证库不仅灵活而且功能丰富。如果仅仅期望理解如何自定义MVC特性以最大限度利用可用的验证选项，这是值得的。*Pro jQuery 2.0*一书深入介绍了jQuery验证库。

27.6 执行远程验证

本章将要描述的最后一个验证特性是远程验证。这是一种客户端验证技术，可以通过调用服务器端的操作方法来执行验证。

远程验证的常见示例是在应用中检查用户名是否可用，这些名称必须是唯一的，用户提交数据，

然后执行客户端验证。作为处理的一部分，向服务器发出Ajax请求以验证请求的用户名。如果用户名已经被占用，则显示验证错误以便用户输入另外的名称。

这可能看起来类似于常规的服务器端验证，但这种方法有一些好处。首先，仅有少量属性被远程验证；客户端验证的好处仍然适用于用户输入的所有其他属性值。其次，请求是轻量级的并专注于验证，而不是处理整个模型对象。

远程验证在后台执行，用户不需要单击提交按钮并等待新的视图被渲染和返回。这使得用户体验更好，尤其是在浏览器和服务器之间网速较慢的时候。

也就是说，远程验证做了妥协，它在客户端验证和服务器端验证之间取得平衡，但它确实需要向

应用服务器发出请求，因而不像通常的客户端验证那样快速。

使用远程验证的第一步是创建一个可以验证模型属性的操作方法，下面将验证Appointment模型的Date属性来确保请求的预约发生在未来（这是本章开头使用的原有验证规则之一，但是不可能使用标准的客户端验证特性来实现）。代码清单27-22展示了如何将操作方法ValidateDate添加到Home控制器中。

代码清单27-22　在Controllers文件夹下的HomeController.cs文件中加入验证用的操作方法

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;
namespace ModelValidation.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() =>
            View("MakeBooking", new Appointment() { Dat
```

```
e = DateTime.Now });

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
 {

            if (ModelState.GetValidationState(nameof(ap
pt.Date))
                    == ModelValidationState.Valid
                && ModelState.GetValidationState(nameof
(appt.ClientName))
                    == ModelValidationState.Valid
                && appt.ClientName.Equals("Joe", String
Comparison.OrdinalIgnoreCase)
                && appt.Date.DayOfWeek == DayOfWeek.Mon
day) {
                ModelState.AddModelError("",
                    "Joe cannot book appointments on Mo
ndays");
            }

            if (ModelState.IsValid) {
                return View("Completed", appt);
            } else {
                return View();
            }
        }

        public JsonResult ValidateDate(string Date) {
            DateTime parsedDate;

            if (!DateTime.TryParse(Date, out parsedDate
)) {
                return Json("Please enter a valid date
(mm/dd/yyyy)");
            } else if (DateTime.Now > parsedDate) {
```

```
                    return Json("Please enter a date in the
 future");
                } else {
                    return Json(true);
                }
            }
        }
}
```

支持远程验证的操作方法必须返回JsonResult类型，以告诉MVC正在使用JSON数据。如第20章所述，除结果之外，验证用的操作方法必须定义与要验证的数据字段同名的参数。对于这个示例来说，在操作方法内部，通过将值解析为DateTime对象并检查是否为未来的时间来执行验证。

提　示

你可以利用模型绑定带来的好处将操作方法的

参数定义为DateTime对象，但是这样做意味着如果用户输入没有意义的值，比如apple，验证方法将无法处理。这是因为模型绑定器无法从apple创建DateTime对象，并在试图处理时抛出异常。远程验证没有办法表达这个异常，因此被无提示地丢弃。这导致不能突出显示数据字段的未预期结果，产生用户输入的数据是有效的印象。作为一般规则，远程验证的最佳方法是在操作方法中接收字符串类型的参数，并执行显式的类型转换、解析或模型绑定。

使用Json方法可以表达验证结果，Json方法创建了客户端远程验证脚本可以解析和处理的JSON格式的结果。如果值是有效的，就将true作为参数传递给Json方法，如下所示：

```
...
return Json(true);
...
```

如果有问题，就将用户应该看到的验证错误信息作为参数传递给Json方法，如下所示：

```
...
return Json("Please enter a date in the future");
...
```

为了使用远程验证方法，可在模型类的属性上应用Remote特性，如代码清单27-23所示。

代码清单27-23　在Models文件夹下的Appointment.cs文件中使用Remote特性

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;
using Microsoft.AspNetCore.Mvc;

namespace ModelValidation.Models {

    public class Appointment {

        [Required]
```

```
        [Display(Name = "name")]
        public string ClientName { get; set; }

        [UIHint("Date")]
        [Required(ErrorMessage = "Please enter a date")
]
        [Remote("ValidateDate", "Home")]
        public DateTime Date { get; set; }
        [MustBeTrue(ErrorMessage = "You must accept the
 terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

Remote特性的参数是用于生成JavaScript验证库将要调用的URL以便执行验证的操作方法名称和控制器名称，在本例中是Home控制器的ValidateDate方法。

启动应用，导航到/Home并输入过去的日期来查看远程验证如何工作。当你选择一个值并将焦点转移到另外一个元素时，将会显示验证消息，如图27-8所示。

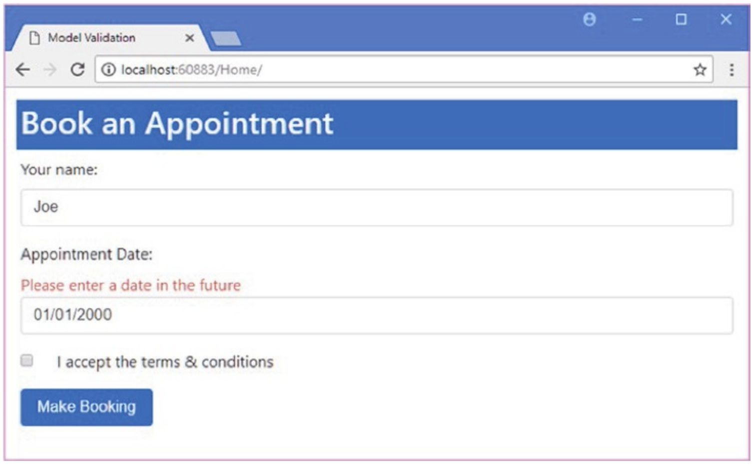


图27-8　执行远程验证



警　告

在用户首次提交表单以及以后每次编辑数据时，都会调用验证用的操作方法。对于文本类型的input元素，每次单击都将导致调用到服务器。对于某些应用，这可能是大量请求，在生产中指定服务器容量和带宽时必须考虑这些请求。此外，可以选

择不对验证成本高昂的属性使用远程验证（例如，当必须查询较慢的服务器以确定用户名是否唯一时）。

27.7　小结

本章说明了可用于执行模型验证的各种技术，确保用户提供的数据与对数据模型的约束一致。模型验证是一个重要的话题，为应用得到正确的验证对于确保用户具有良好和无挫败感的体验至关重要。下一章将说明如何使用ASP.NET Core Identity来保护MVC应用程序。

第28章　ASP.NET Core Identity入门

ASP.NET Core Identity是来自微软的用于在ASP.NET应用程序中管理用户的API。本章将演示它的设置过程，并创建一个简单的用户管理工具来管理存储在数据库中的独立用户账户。

ASP.NET Core Identity也支持其他类型的用户账户，比如存储在活动目录中的用户账户，但不会描述它们，因为它们不经常用于企业之外的环境（活动目录的实现往往很复杂，很难提供有用的通用示例）。

注　意

本章要求为Visual Studio安装SQL Server LocalDB功能。可以通过运行Visual Studio安装器和安装Microsoft SQL Server Data Tools来添加SQL Server LocalDB。

第29章将展示如何使用这些用户账户执行身份验证和授权，第30章将展示如何在这些基础之上应用一些高级技术。表28-1提供了关于ASP.NET Core Identity的问题。

表**28-1**　关于**ASP.NET Core Identity**的问题

问题	答案
是什么？	ASP.NET Core Identity是管理用户并通过Entity Framework Core将用户数据存储到诸如数据库的存储库中的API

有何作用？	对于大多数应用程序来说，用户管理是重要的功能。ASP.NET Core Identity提供了现成的和经过良好测试的平台，对于常见功能甚至不需要创建自定义的版本
如何使用？	可通过添加到Startup类的服务和中间件，以及作为应用和ASP.NET Core Identity之间桥梁的类来使用
有何缺陷或限制？	ASP.NET Core Identity更加灵活且可配置，能够弥补早期的ASP.NET用户管理API的不足
有何替代方案？	可以实现自己的API，但是需要做大量的工作，而且除非非常小心，否则可能导致安全漏洞

表28-2列出了本章要介绍的操作。

表**28-2**　本章要介绍的操作

操作	方法	代码清单
为项目添加ASP.NET Core Identity	为ASP.NET Core Identity和Entity Framework Core添加包和中间件，创建用户类和数据库上下文类，创建数据库迁移	代码清单28-1～代码清单28-13
读取用户数据	使用上下文类查询ASP.NET Core Identity数据库	代码清单28-14和代码清单28-15

创建用户账户	调用UserManager.CreateAsync方法	代码清单28-16～代码清单28-18
创建默认密码策略	在Startup类中设置密码配置	代码清单28-19
实现自定义密码验证	实现IPasswordValidator接口或者从PasswordValidator派生子类	代码清单28-20～代码清单28-22
改变账户验证策略	在Startup类中设置用户配置	代码清单28-23
实现自定义账户验证	实现IUserValidator接口或者从UserValidator派生子类	代码清单28-24～代码清单28-26
删除用户账户	调用UserManager.DeleteAsync方法	代码清单28-27和代码清单28-28
编辑用户账户	调用UserManager.UpdateAsync方法	代码清单28-29～代码清单28-31

28.1　准备示例项目

在本章中，使用ASP.NET Core Web Application (.NET Core)模板创建新的名为Users的Empty项目。这个示例应用需要Entity Framework Core命令行工具，但该工具必须通过手动编辑.csproj文件来加入项目。在Solution Explorer窗格中右击Users项目，从弹出的菜单中选择Edit Users.csproj文件，并添加代码清单28-1所示的元素。

代码清单28-1　在Users文件夹下的Users.csproj文件中添加包

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All
" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFr
ameworkCore.Tools.DotNet"
```

```
        Version="2.0.0" />
  </ItemGroup>

</Project>
```

代码清单28-2展示了Startup类，在其中配置MVC框架并启用开发中要用到的中间件。

代码清单28-2　Users文件夹下的Startup.cs文件的内容

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace Users {

    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app) {

            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

创建控制器和视图

创建Controllers文件夹，添加名为HomeController.cs的类文件，定义代码清单28-3所示的控制器。后面将使用Home控制器来详尽描述用户账户和数据，操作方法Index通过View方法传递了一个字典给默认视图。

代码清单28-3　Controllers文件夹下的HomeController.cs文件的内容

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() =>
            View(new Dictionary<string, object>
                {["Placeholder"] = "Placeholder" });
    }
}
```

为了给控制器提供视图，创建Views/Home文

件夹，添加名为Index.cshtml的视图文件，内容如代码清单28-4所示。

代码清单28-4　Views/Home文件夹下的Index.cshtml文件的内容

```
@model Dictionary<string, object>

<div class="bg-primary m-1 p-1 text-white"><h4>User Det
ails</h4></div>

<table class="table table-sm table-bordered m-1 p-1">
    @foreach (var kvp in Model) {
        <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
    }
</table>
```

Index视图使用表格来显示模型字典的内容。为支持视图，创建Views/Shared文件夹，添加名为_Layout.cshtml的视图文件，内容如代码清单28-5所示。

代码清单28-5　Views/Shared文件夹下的_Layout.cshtml文件的内容

```
<!DOCTYPE html>
<html>
<head>
    <title>Users</title>
    <meta name="viewport" content="width=device-width"
/>
    <link href="/lib/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

这个视图基于Bootstrap CSS包来修饰HTML元素。为了将Bootstrap包添加到项目中，可使用Bower Configuration File模板创建bower.json文件，然后在dependencies部分添加Bootstrap包，如代码清单28-6所示。

代码清单28-6　在Users文件夹下的bower.json文件中添加Bootstrap包

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
```

```
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

要做的最后的准备工作是在Views文件夹中创建_ViewImports.cshtml文件，以设置视图中使用的内置标签助手，如代码清单28-7所示。

代码清单28-7　Views文件夹下的_ViewImports.cshtml文件的内容

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

在Views文件夹中添加名为_ViewStart.cshtml的视图文件，内容如代码清单28-8所示，用以确保你在代码清单28-5中创建的布局将被用于应用的所有视图。

代码清单28-8　Views文件夹下的_ViewStart.cshtml文件的内容

```
@{
    Layout = "_Layout";
}
```

执行示例应用，你将看到图28-1所示的输出。

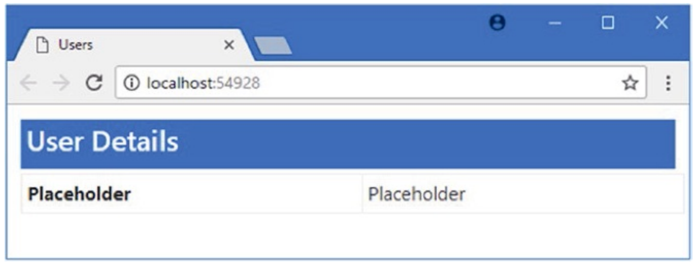


图28-1　输出

28.2　设置ASP.NET Core Identity

设置ASP.NET Core Identity的过程几乎涉及应用的各个部分，并且需要新的模型类、配置更新、控制器和操作方法来支持验证和授权操作。下面将介绍ASP.NET Core Identity的基本配置以展示涉及的不同步骤。有多种不同的方式在应用中使用ASP.NET Core Identity，本章遵循最简单、最常用的配置。

28.2.1　创建用户类

第一步是定义用来表示应用中用户的类，称为用户类。

用户类派生自IdentityUser类，该类定义在Microsoft.AspNetCore.Identity.EntityFrameworkCore命名空间中。IdentityUser类提供基本的用户表示，可以通过在派生类中添加属性来扩展，详见第30章。表28-3展示了IdentityUser类定义的属性。

表**28-3　IdentityUser**类定义的属性

属性	描述
Id	包含用户的唯一ID
UserName	返回用户的名字
Claims	返回用户的凭据集合
Email	包含用户的电子邮件地址
Logins	返回用户的登录集合，用于第三方验证

PasswordHash	返回用户密码的散列值
Roles	返回用户所属的角色集合
PhoneNumber	返回用户的电话号码
SecurityStamp	当用户的身份发生变化之后，返回新的值，例如当密码发生变更之后

单独的属性此刻并不重要，重要的是，IdentityUser类可以访问用户的基本信息——用户名、电子邮件、电话号码、密码的散列值、成员角色以及其他信息。如果期望保存用户的额外信息，就必须在从IdentityUser类派生的类中添加属性，从而在应用中用来表示用户。

为了在应用中创建用户，创建Models文件夹，并添加名为AppUser.cs的类文件来创建AppUser类，如代码清单28-9所示。

代码清单28-9　Models文件夹下的AppUser.cs文件的内容

```
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

    public class AppUser : IdentityUser {
        // no additional members are required
        // for basic Identity installation
    }
}
```

这就是此刻必须要做的事性，第30章在展示如何添加特定于应用的用户数据属性时，还会讨论这个类。

配置视图导入

虽然与设置ASP.NET Core Identity没有直接关系，但仍需要在视图中使用AppUser对象。为了简化视图的创建，可在视图导入文件中添加Users.Models命名空间，如代码清单28-10所示。

代码清单28-10　在Views文件夹下的_ViewImports.cshtml文件中添加命名空间

```
@using Users.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

28.2.2　创建数据库上下文类

下一步是创建操作AppUser类的Entity Framework Core数据库上下文类。数据库上下文类派生自IdentityDbContext<T>，这里的T就是用户类（示例项目中的AppUser类）。添加名为AppIdentityDbContext.cs的类文件到Models文件夹中，并定义代码清单28-11所示的类。

代码清单28-11　Models文件夹下的AppIdentityDbContext.cs文件的内容

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore
;
using Microsoft.EntityFrameworkCore;

namespace Users.Models {

    public class AppIdentityDbContext : IdentityDbConte
xt<AppUser> {

        public AppIdentityDbContext(DbContextOptions<Ap
```

```
pIdentityDbContext> options)
            : base(options) { }
    }
}
```

数据库上下文类可以扩展为自定义数据库的设置和使用方式，但是对于基本的ASP.NET Core Identity应用来说，仅仅定义这个类即可开始使用，并为将来的任何定制提供扩展空间。

注　意

不必担心这些类是否有意义。如果不熟悉Entity Framework Core，建议将其视为黑盒。一旦基本的构件块就绪，就可以将它们复制到项目中并使用，几乎不需要修改它们。

28.2.3　配置数据库连接串

ASP.NET Core Identity的第一个配置步骤是定义用于数据库的数据库连接串。默认会将数据库连接串放在appsettings.json文件中，在应用启动时，该文件将由Startup类加载并由Startup类访问。使用ASP.NET Configuration File模板在项目的根目录中创建appsettings.json文件，并添加代码清单28-12所示的配置。

代码清单28-12　Users文件夹下的appsettings.json文件的内容

```
{
  "Data": {
    "SportStoreIdentity": {
      "ConnectionString":
  "Server=(localdb)\\MSSQLLocalDB;Database=IdentityUser
s;Trusted_Connection=True;
  MultipleActiveResultSets=true"
    }
  }
}
```

在数据库连接串中指定localdb选项，从而为开发人员提供方便的数据库支持。对于数据库本身，可指定名称IdentityUsers。

注　意

数据库连接串必须在一行中，这在Visual Studio中效果很好，但是对于印刷而言，受限于纸张大小，不得不分多行显示。

使用数据库连接串，可以更新Startup类来读取配置文件并使配置可用，如代码清单28-13所示。

代码清单28-13　在Users文件夹下的Startup.cs文件中读取应用

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;

namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
()
                .AddEntityFrameworkStores<AppIdentityDb
Context>()
                .AddDefaultTokenProviders();

            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

创建基本的ASP.NET Core Identity安装需要3组更新。第一组是设置Entity Framework (EF) Core，从而为MVC应用提供数据访问服务。

```
...
services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(Configuration["Data:SportStore
Identity:ConnectionString"]));
...
```

AddDbContext方法用于添加Entity Framework Core所需的服务，UseSqlServer方法用于设置使用Microsoft SQL Server存储数据所需的支持。AddDbContext方法允许你应用之前创建的数据库上

下文类，使用从应用配置中获取的数据库连接串中的SQL Server数据库进行备份（对于示例应用，这意味着读取appsettings.json文件）。

你还需要设置ASP.NET Core Identity服务，如下所示：

```
...
services.AddIdentity<AppUser, IdentityRole>()
    .AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

AddIdentity方法的类型参数用来指定表示用户和角色的类。以上代码为用户指定了AppUser类，为角色指定了内置的IdentityRole类。使用之前创建的数据库上下文类，AddEntityFrameworkStores方法指定了应当使用Entity Framework Core来存储和检索数据。AddDefaultTokenProviders方法用于使用默认配置来支持获取token，例如修改口令。

对Startup类所做的最终修改是将ASP.NET Core Identity添加到请求处理管线中，这意味着允许用户凭据与基于Cookie或URL重写的请求相关联，还意味着用户账户的详细内容不会直接包含在发送给应用的HTTP请求或生成的响应中。

```
...
app.UseAuthentication();
...
```

28.2.4 创建ASP.NET Core Identity数据库

一切几乎就绪，仅剩的步骤是实际创建用来存储ASP.NET Core Identity数据的数据库。打开命令行窗口或PowerShell窗口，导航到Users文件夹（包含Startup.cs类文件的那个），然后执行如下命令：

```
dotnet ef migrations add Initial
```

正如为SportsStore应用设置数据库时说明的那

样，Entity Framework Core通过名为迁移的特性管理对数据库模式所做的更新。当修改用于生成架构的模型类时，可以生成包含SQL命令的迁移文件来更新数据库。该命令将创建为ASP.NET Core Identity设置数据库的迁移文件。

当执行完这个命令之后，你将在Solution Explorer窗格中看到Migrations文件夹。检查其中的内容，你就可以看到用来创建初始数据库的SQL命令。为了使用这些迁移文件来创建数据库，可运行如下命令：

```
dotnet ef database update
```

完成这个过程可能需要一点时间，一旦命令完成，数据库将被创建并备用。

28.3　使用ASP.NET Core Identity

现在，基本的设置已经完成，可以开始使用ASP.NET Core Identity为示例应用增加用户管理支持了。下一步将展示ASP.NET Core Identity API如何被用来创建集中管理用户的管理工具。

集中化的用户管理工具在所有应用中都很有用，甚至可以允许用户创建和管理自己的账户。总有一些客户需要批量创建账户，比如，需要检查和调整用户数据的支持问题。管理工具非常有用，因为它们将很多基本的用户管理功能整合到了少量的类中，使之成为演示ASP.NET Core Identity基本功能的有用示例。

28.3.1　列举用户账户

本节的起点是列举数据库中所有的用户账户，这可以使你看到稍后添加到应用中的代码的效果。Controllers文件夹中添加名为AdminController.cs的

类文件，并定义代码清单28-14所示的控制器，进而定义用户管理功能。

代码清单28-14　Controllers文件夹下的AdminController.cs文件的内容

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {
    public class AdminController : Controller {
        private UserManager<AppUser> userManager;

        public AdminController(UserManager<AppUser> usr
Mgr) {
            userManager = usrMgr;
        }

        public ViewResult Index() => View(userManager.U
sers);
    }
}
```

操作方法Index将枚举ASP.NET Core Identity系统管理的用户。当然，现在没有任何用户，但很快就会有。可通过控制器类的构造函数接收的

UserManager<AppUser>对象来访问用户数据，该对象由依赖注入系统提供。

有了UserManager<AppUser>对象，便可以开始查询数据存储。Users属性将返回用户对象的枚举值——在应用中就是AppUser实例——可以使用LINQ进行查询和操控。在操作方法中，将Users属性传递给View方法，以便我们可以显示账户的详情。为了给操作方法提供视图，创建Views/Admin文件夹，并添加名为Index.cshtml的视图文件，应用代码清单28-15所示的标记内容。

代码清单28-15　Views/Admin文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<AppUser>

<div class="bg-primary m-1 p-1 text-white"><h4>User Accounts</h4></div>

<table class="table table-sm table-bordered">
    <tr><th>ID</th><th>Name</th><th>Email</th></tr>
    @if (Model.Count() == 0) {
```

```
        <tr><td colspan="3" class="text-center">No User
 Accounts</td></tr>
    } else {
        foreach (AppUser user in Model) {
            <tr>
                <td>@user.Id</td>
                <td>@user.UserName</td>
                <td>@user.Email</td>
            </tr>
        }
    }
</table>

<a class="btn btn-primary" asp-action="Create">Create</
a>
```

Index视图包含一个表格，其中的每一行对应一个用户，表格的列则包括唯一ID、用户名和电子邮件地址。如果数据库中没有用户，那么会显示一条消息，如图 28-2 所示，启动应用并访问/Admin就可以看到。

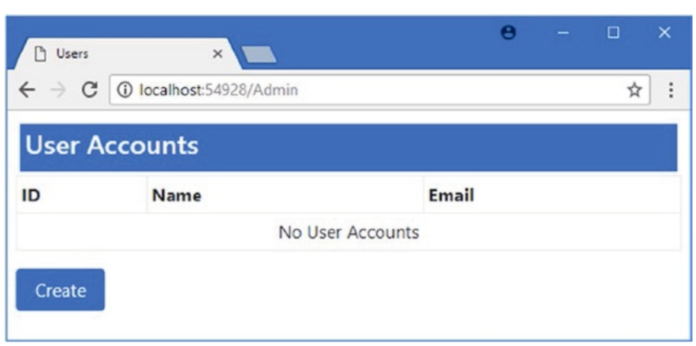


图28-2　显示用户列表（目前是空的）

Index视图中包含了按钮样式的Create锚点元素，目标为Admin控制器的Create方法，用来支持添加用户的操作。

重置数据库

可以通过打开Visual Studio的SQL Server对象资源管理器来查看创建的数据库。如果这是你第一次使用SQL Server对象资源管理器，那么需要从Tools菜单中选择Connect to Database来告诉Visual Studio要使用的数据库。对于数据源，选择Microsoft SQL Server，使用(localdb)\mssqllocaldb作为服务名，选择使用Windows身份验证，单击Select Or Enter a Database Name字段的下拉箭头。几秒后，你将看到可用的LocalDB数据库列表，你应该可以选择

IdentityUsers，这是示例应用的数据库。单击OK按钮，SQL Server Object Explorer窗口中会显示一个新的条目。Visual Studio将记住该数据库，因此只需要执行这个过程一次。

在SQL Server Object Explorer窗口中，选择(localdb)\mssqllocaldb→Databases→IdentityUsers，浏览数据库。你将看到由迁移文件创建的表，名称类似于AspNetUsers和AspNetRoles。一旦将用户添加到数据库中，就可以查询数据库以便查看表的内容。

要删除数据库，可右击SQL Server Object Explorer窗口中的IdentityUsers项目，然后从弹出的菜单中选择Delete，在数据库删除对话框中选择所有的选项，然后单击OK按钮删除。

要重新创建数据库，可打开Package Manager

Console窗口，并运行如下命令：

```
dotnet ef database update
```

数据库将被重建，并在下次启动应用时可用。

28.3.2　创建用户

要为应用接收的输入应用MVC模型验证，最简单的方式是为控制器支持的每个操作创建简单的视图模型。添加名为UserViewModels.cs的类文件到Models文件夹中，并用它定义代码清单28-16所示的类。

代码清单28-16　Models文件夹下的UserViewModels.cs文件的内容

```
using System.ComponentModel.DataAnnotations;

namespace Users.Models {
```

```
    public class CreateModel {
        [Required]
        public string Name { get; set; }
        [Required]
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
    }
}
```

定义的初始模型名为CreateModel，它定义了创建用户账户所需的基本属性——用户名、电子邮件地址和密码。可使用来自System.ComponentModel.DataAnnotations命名空间的Required特性来表示定义在模型中的3个属性是必需的。

代码清单28-17为Admin控制器添加了一对Create操作方法，它们可由Index视图中的链接定位，并使用标准的控制器模式对GET请求向用户呈现视图，使用POST请求处理表单数据。

代码清单28-17　在Controllers文件夹下的AdminController.cs文件中定义Create操作方法

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {

    public class AdminController : Controller {
        private UserManager<AppUser> userManager;

        public AdminController(UserManager<AppUser> usr
Mgr) {
            userManager = usrMgr;
        }

        public ViewResult Index() => View(userManager.U
sers);

        public ViewResult Create() => View();

        [HttpPost]
        public async Task<IActionResult> Create(CreateM
odel model) {
            if (ModelState.IsValid) {
                AppUser user = new AppUser {
                    UserName = model.Name,
                    Email = model.Email
                };
                IdentityResult result
                    = await userManager.CreateAsync(use
r, model.Password);
```

```
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    foreach (IdentityError error in res
ult.Errors) {
                        ModelState.AddModelError("", er
ror.Description);
                    }
                }
            }
            return View(model);
        }
    }
}
```

以上代码中的Create操作方法接收一个CreateModel参数，它将在管理员提交表单数据时被调用。ModelState.IsValid属性用于检查数据是否包含所需的值，如果包含，则创建AppUser类的新实例并传递给异步的UserManager.CreateAsync方法，如下所示：

```
...
AppUser user = new AppUser { UserName = model.Name, Ema
il = model.Email };
IdentityResult result = await userManager.CreateAsync(u
ser, model.Password);
```

```
...
```

CreateAsync方法的返回结果是一个IdentityResult对象，可通过表28-4所示的属性来描述操作结果。

表**28-4　IdentityResult**类定义的属性

属性	描述
Succeeded	如果操作成功，返回true
Errors	返回用于描述在尝试操作时遇到的错误的IdentityError对象序列。IdentityError类提供了Description属性来汇总问题

在Create操作方法中检查Succeeded属性，以确定是否在数据库中创建了新的用户。如果Succeeded属性为true，客户将被重定向到Index操作以便用户列表显示出来。

```
...
if (result.Succeeded) {
    return RedirectToAction("Index");
```

```
} else {
    foreach (IdentityError error in result.Errors) {
        ModelState.AddModelError("", error.Description)
;
    }
}
...
```

如果Succeeded属性为false，那么Errors属性提供的IdentityError对象序列将被枚举，Description属性将被ModelState.AddModelError方法用来创建模型级验证错误，如第27章所述。

为了给新的操作方法提供视图，在Views/Admin文件夹中创建名为Create.cshtml的视图文件，并添加代码清单28-18所示的标记内容。

代码清单28-18　Views/Admin文件夹下的Create.cshtml文件的内容

```
@model CreateModel

<div class="bg-primary m-1 p-1 text-white"><h4>Create U
ser</h4></div>
<div asp-validation-summary=" All" class="text-danger">
```

```
</div>
<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control"
/>
    </div>
    <button type="submit" class="btn btn-primary">Creat
e</button>
    <a asp-action="Index" class="btn btn-secondary">Can
cel</a>
</form>
```

Create视图没什么特别之处，只是一个简单的表单，用于收集MVC用来绑定到模型类的属性的值，模型类的属性被传递给Create操作方法，并包含有验证错误时的摘要。

测试创建功能

为了测试创建新用户账户的能力，启动应用并导航到URL地址/Admin，然后单击Create按钮，使用表28-5提供的值填写表单。

提　示

人们为测试保留了一些域名，如example.com。你可以在IETF网站上看到完整列表。

表**28-5**　创建示例用户的值（一）

名字	值
Name	Joe

Email	joe@example.com
Password	Secret123$

输入值之后，单击Create按钮，ASP.NET Core Identity将会创建用户账户。当浏览器被重定向到Index操作方法之后，将显示创建的用户账户，如图28-3所示（你将会看到不一样的ID，因为每个账户会随机生成ID）。

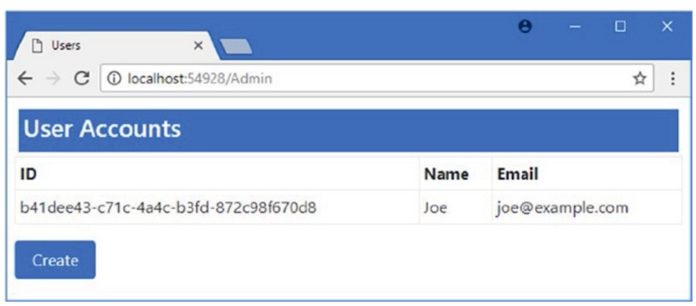


图28-3　添加新的用户账户

再次单击Create按钮并在表单中输入表28-5中同样的值。提交表单后，你将看到由模型验证摘要报告的错误，如图28-4所示。

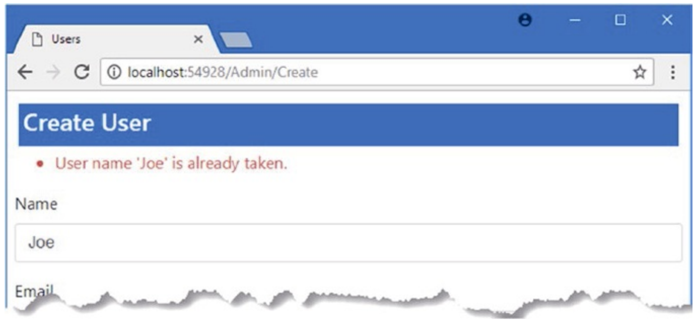


图28-4　因试图创建相同用户账户产生的错误

28.3.3　验证密码

对于企业级应用，最常见的需求之一是强制密码策略。可以通过运行应用来查看默认的策略，访问/Admin/Create，然后使用表28-6所示的数据填充表单，与之前的重要区别在于Password字段。

表**28-6**　创建示例用户的值（二）

名称	值
Name	Alice
Email	alice@example.com
Password	secret

当提交表单到服务器之后，ASP.NET Core Identity系统检查备选的密码，如果不符合要求，将生成错误消息，生成的错误消息如图28-5所示。

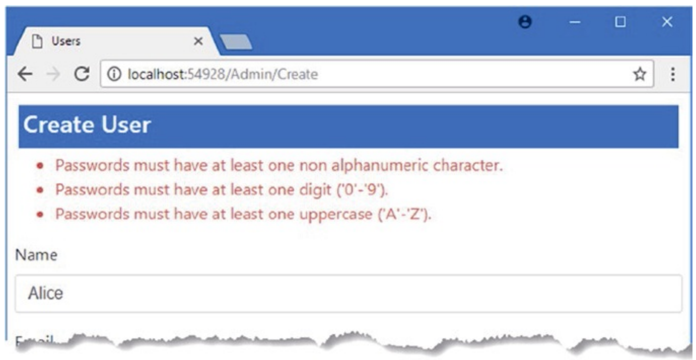


图28-5　密码验证错误消息

可以在Startup类中配置密码验证规则，如代码清单28-19所示。

代码清单28-19　在Users文件夹下的Startup.cs文件中配置密码验证规则

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
```

```
using Users.Models;
namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
(opts => {
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
                .AddDefaultTokenProviders();

            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
```

```
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

AddIdentity方法可以与一个接收IdentityOptions对象的函数一起使用，该函数的Password属性将返回一个PasswordOptions实例，PasswordOptions类提供了如表28-7所示的用于管理密码策略的属性。

表**28-7　PasswordOptions**类定义的属性

属性	描述
RequiredLength	用于指定密码的最小长度
RequireNonAlphanumeric	为true时要求密码至少包含一个非字符或数字的字符
RequireLowercase	为true时要求密码至少包含一个小写字母
RequireUppercase	为true时要求密码至少包含一个大写字母

RequireDigit	为true时要求密码至少包含一个数字

启动应用，导航到/Admin/Create，然后重新提交表单，你将看到密码secret被接受，并为Alice创建了一个新的账户，如图28-6所示。

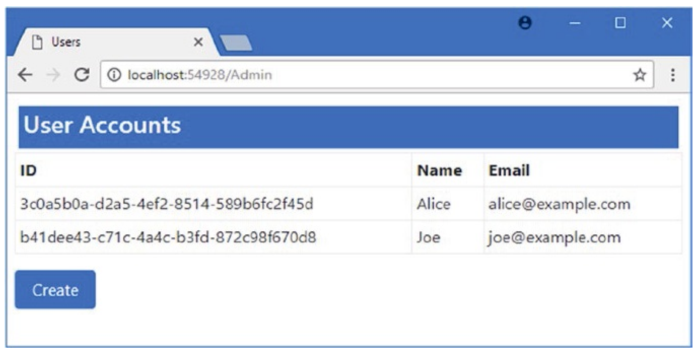


图28-6　改变密码验证策略

实现自定义的密码验证策略

内置的密码验证策略对于大多数应用是足够的，但是你可能需要实现自定义的密码验证策略，特别是当你正在为企业级在线业务应用实施常见的复杂密码验证策略时。密码验证功能由Microsoft.AspNetCore.Identity命名空间的

IPasswordValidator<T>接口定义，这里的T是应用特定的用户类（在示例中是AppUser）。

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {

    public interface IPasswordValidator<TUser> where TUser : class {

        Task<IdentityResult> ValidateAsync(UserManager<TUser> manager,
            TUser user, string password);
    }
}
```

ValidateAsync方法被调用以验证密码，UserManager对象提供了数据上下文（允许查询ASP.NET Core Identity数据库），该对象表示用户和候选密码。返回的结果是一个IdentityResult对象，如果验证没有问题，则使用静态的Success属性创建这个对象；否则，使用静态的Failed方法，该方法会传递一个IdentityError数组对象，其中的每个对象描述一个验证问题。

为了演示自定义的密码验证策略，创建Infrastructure文件夹，添加名为CustomPasswordValidator.cs的类文件，并定义代码清单28-20所示的类。

代码清单28-20　Infrastructure文件夹下的CustomPasswordValidator.cs文件的内容

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {

    public class CustomPasswordValidator : IPasswordVal
idator<AppUser> {

        public Task<IdentityResult> ValidateAsync(UserM
anager<AppUser> manager,
                AppUser user, string password) {

            List<IdentityError> errors = new List<Ident
ityError>();

            if (password.ToLower().Contains(user.UserNa
me.ToLower())) {
                errors.Add(new IdentityError {
                    Code = "PasswordContainsUserName",
```

```
                    Description = "Password cannot cont
ain username"
                });
            }
            if (password.Contains("12345")) {
                errors.Add(new IdentityError {
                    Code = "PasswordContainsSequence",
                    Description = "Password cannot cont
ain numeric sequence"
                });
            }

            return Task.FromResult(errors.Count == 0 ?
                IdentityResult.Success : IdentityResult
.Failed(errors.ToArray()));
        }
    }
}
```

验证器类检查密码并确保不包含用户名和序列12345。代码清单28-21为AppUser对象注册了CustomPasswordValidator类作为密码验证器。

代码清单28-21　在Startup.cs文件中注册自定义的密码验证器

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
```

```
using Users.Infrastructure;

namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddTransient<IPasswordValidator<Ap
pUser>,
                CustomPasswordValidator>();

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
(opts => {
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
                .AddDefaultTokenProviders();
```

```
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

为了测试自定义的密码验证策略，启动应用并访问/Admin/Create，使用表28-8所示的数据填充表单。

表**28-8**　创建示例用户的值（三）

名称	值
Name	Bob
Email	bob@example.com
Password	bob12345

表28-8中的密码违反了两个验证规则，因而返回图28-7所示的错误消息。

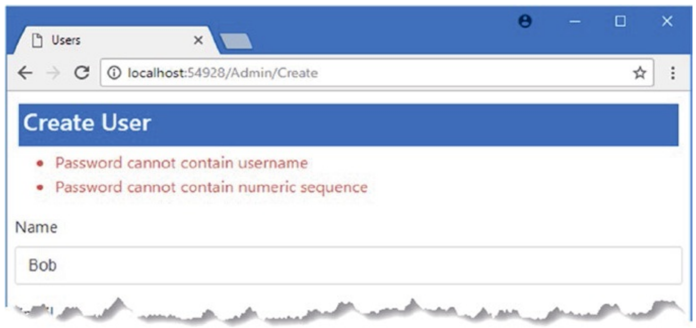


图28-7　使用自定义的密码验证器

也可以基于默认的内置验证器来实施自定义的验证规则。默认的验证器为定义在Microsoft.AspNetCore.Identity命名空间中的PasswordValidator。代码清单28-22将自定义的验证器修改为派生自PasswordValidator类，并基于提供的基本检查来实现验证。

代码清单28-22　在CustomPasswordValidator.cs文件中从内置的验证器派生自定义的验证器

```
using System.Collections.Generic;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
using System.Linq;

namespace Users.Infrastructure {

    public class CustomPasswordValidator : PasswordVali
dator<AppUser> {
        public override async Task<IdentityResult> Vali
dateAsync(
                    UserManager<AppUser> manager, AppUs
er user, string password) {

            IdentityResult result = await base.Validate
Async(manager,
                user, password);

            List<IdentityError> errors = result.Succeed
ed ?
                new List<IdentityError>() : result.Erro
rs.ToList();

            if (password.ToLower().Contains(user.UserNa
me.ToLower())) {
                errors.Add(new IdentityError {
                    Code = "PasswordContainsUserName",
                    Description = "Password cannot cont
ain username"
                });
            }
            if (password.Contains("12345")) {
                errors.Add(new IdentityError {
                    Code = "PasswordContainsSequence",
                    Description = "Password cannot cont
ain numeric sequence"
```

```
                });
            }

            return errors.Count == 0 ? IdentityResult.S
uccess
                : IdentityResult.Failed(errors.ToArray(
));
        }
    }
}
```

为了测试组合验证，可运行应用并在/Admin/Create返回的表单中使用表28-9所示的数据填充表单。

表**28-9**　创建示例用户的值（四）

名称	值
Name	Bob
Email	bob@example.com
Password	12345

提交表单后，你将看到组合了自定义验证和内

置验证的错误消息，如图28-8所示。

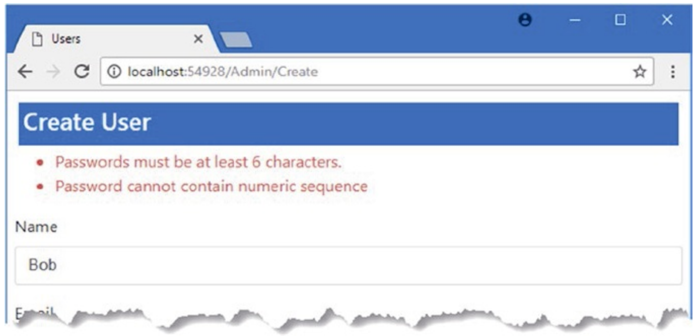


图28-8　组合自定义验证和内置验证

28.3.4　验证用户详情

在创建账户的时候，验证也同样作用于用户名和电子邮件地址。为了查看内置验证，启动应用并使用表28-10所示的数据填充地址/Admin/Create返回的表单。

表**28-10**　创建示例用户的值（五）

名称	值
Name	Bob!

Email	alice@example.com
Password	secret

提交表单后，你将看到图28-9所示的验证错误消息。



图28-9　验证错误消息

验证可以在Startup类中使用IdentityOptions.User属性进行配置，该属性将返回UserOptions类的实例，表28-11描述了UserOptions类定义的属性。

表**28-11　UserOptions**类定义的属性

属性	描述

AllowedUserNameCharacters	包含可以用于用户名的所有字符，默认值为a～z、A～Z和0～9以及连字符（-）、句点（.）、下画线（_）和@。这个属性不是正则表达式，并且任何合法字符必须显式包含在这个字符串中
RequireUniqueEmail	为true表示要求新账户的电子邮件地址尚未使用

代码清单28-23修改了应用的配置以便要求唯一的电子邮件地址，以及仅有小写字母可用于用户名。

代码清单28-23　在Startup.cs文件中修改用户账户的设置

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;

namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;
```

```
        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddTransient<IPasswordValidator<Ap
pUser>,
                CustomPasswordValidator>();

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
(opts => {
              opts.User.RequireUniqueEmail = true;
              opts.User.AllowedUserNameCharacters = "ab
cdefghijklmnopqrstuvwxyz";
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
                .AddDefaultTokenProviders();

            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
```

```
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

重新提交测试的上一个表单，你将会看到电子邮件地址导致新的错误，而且用于用户名的字符仍然被拒绝，如图28-10所示。

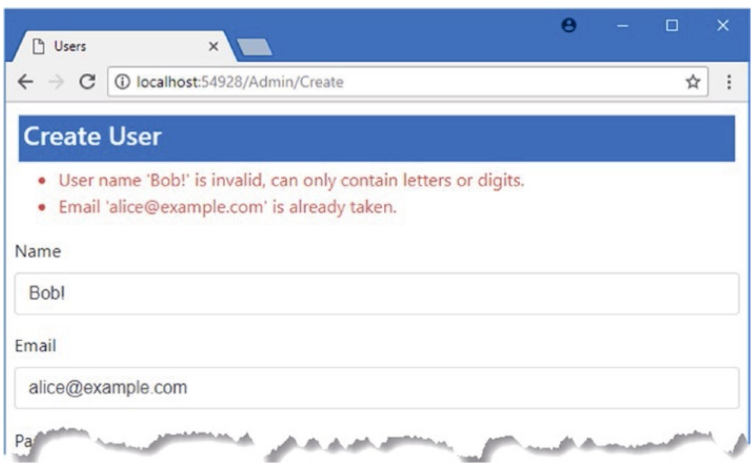


图28-10　修改账户验证设置

实现自定义用户验证

验证功能由接口IUserValidator<T>定义，它定

义在命名空间Microsoft.AspNetCore.Identity中。

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {

    public interface IUserValidator<TUser> where TUser
: class {
        Task<IdentityResult> ValidateAsync(UserManager<
TUser> manager, TUser user);
    }
}
```

ValidateAsync方法被调用以执行验证。返回结果是一个IdentityResult对象，与验证密码的类相同。为了演示自定义验证，在Infrastructure文件夹中添加名为CustomUserValidator.cs的类文件，并用它定义代码清单28-24所示的类。

代码清单28-24　Infrastructure文件夹下的CustomUserValidator.cs文件的内容

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
```

```
namespace Users.Infrastructure {

    public class CustomUserValidator : IUserValidator<A
ppUser> {

        public Task<IdentityResult> ValidateAsync(UserM
anager<AppUser> manager,
                AppUser user) {

            if (user.Email.ToLower().EndsWith("@example
.com")) {
                return Task.FromResult(IdentityResult.S
uccess);
            } else {
                return Task.FromResult(IdentityResult.F
ailed(new IdentityError {
                    Code = "EmailDomainError",
                    Description = "Only example.com ema
il addresses are allowed"
                }));
            }
        }
    }
}
```

上述验证器检查电子邮件的域以确认是example.com域的一部分，代码清单28-25对这个自定义类（作为AppUser对象的验证器）进行了注册。

代码清单28-25　在Startup.cs文件中注册自定义的用户验证器

```
...
public void ConfigureServices(IServiceCollection servic
es) {

    services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
    services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();

    services.AddDbContext<AppIdentityDbContext>(options
 =>
        options.UseSqlServer(
            Configuration["Data:SportStoreIdentity:Conn
ectionString"]));

    services.AddIdentity<AppUser, IdentityRole>(opts =>
 {
        opts.User.RequireUniqueEmail = true;
        opts.User.AllowedUserNameCharacters = "abcdefgh
ijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
    }).AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
...
```

为了测试自定义验证器，运行应用并使用表28-12所示的数据填充地址/Admin/Create返回的表单。

表**28-12**　创建示例用户的值（六）

名字	值
Name	bob
Email	bob@invalid.com
Password	secret

用户名和密码已通过验证，但电子邮件地址不是正确的域。提交表单后，你将看到图28-11所示的错误消息。

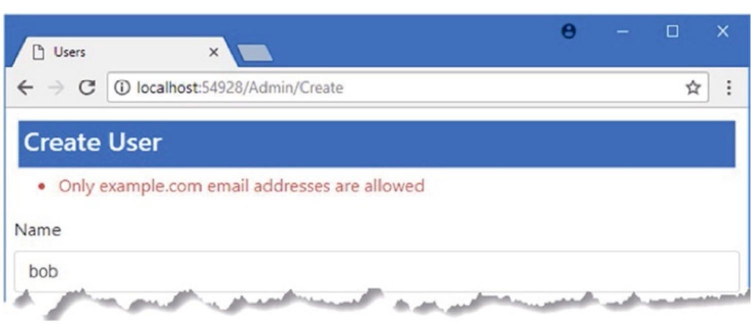


图28-11　错误消息

将UserValidator<T>类提供的内置验证与自定义验证相结合的处理过程与验证密码遵循相同的模式，如代码清单28-26所示。

代码清单28-26　在CustomUserValidator.cs文件中扩展内置的用户验证

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {

    public class CustomUserValidator : UserValidator<AppUser> {

        public override async Task<IdentityResult> ValidateAsync(
                UserManager<AppUser> manager,
                AppUser user) {

            IdentityResult result = await base.ValidateAsync(manager, user);

            List<IdentityError> errors = result.Succeed
```

```
ed ?
                new List<IdentityError>() : result.Erro
rs.ToList();

            if (!user.Email.ToLower().EndsWith("@exampl
e.com")) {
                errors.Add(new IdentityError {
                    Code = "EmailDomainError",
                    Description = "Only example.com ema
il addresses are allowed"
                });
            }

            return errors.Count == 0 ? IdentityResult.S
uccess
                : IdentityResult.Failed(errors.ToArray(
));
        }
    }
}
```

28.4　完成管理功能

只需要实现编辑和删除用户的功能即可完成管理工具。在代码清单28-27中，可以看到对Views/Admin/Index. cshtml文件所做的修改，以便在管理控制台中处理Edit和Delete操作。

代码清单28-27　在Views/Admin文件夹下的Index.cshtml文件中添加Edit和Delete按钮

```
@model IEnumerable<AppUser>

<div class="bg-primary m-1 p-1 text-white"><h4>User Acc
ounts</h4></div>

<div class="text-danger" asp-validation-summary="ModelO
nly"></div>

<table class="table table-sm table-bordered">
    <tr><th>ID</th><th>Name</th><th>Email</th></tr>
    @if (Model.Count() == 0) {
        <tr><td colspan="3" class="text-center">No User
 Accounts</td></tr>
    } else {
        foreach (AppUser user in Model) {
            <tr>
                <td>@user.Id</td><td>@user.UserName</td
><td>@user.Email</td>
                <td>
                    <form asp-action="Delete" asp-route
-id="@user.Id" method="post">
                        <a class="btn btn-sm btn-primar
y" asp-action="Edit"
                            asp-route-id="@user.Id">Edi
t</a>
                        <button type="submit"
                           class="btn btn-sm btn-danger
">Delete</button>
                    </form>
                </td>
            </tr>
```

```
        }
    }
</table>
<a class="btn btn-primary" asp-action="Create">Create</
a>
```

Delete按钮将表单发送到Admin控制器的Delete操作方法，这很重要，因为修改应用状态需要POST请求。Edit按钮是锚点元素，用于发送GET请求，因为编辑过程的第一步是显示当前的数据。Edit按钮包含在表单元素内以便Bootstrap CSS样式不会将其垂直堆叠起来。加入模型验证的摘要信息到视图中，以便可以轻松地显示由其他管理功能导致的错误消息。

28.4.1　实现删除功能

UserManager<T>类定义了DeleteAsync方法，该方法接收AppUser类的实例，并将其从数据库中删除。在代码清单28-28中，可以看到如何使用DeleteAsync方法在Admin控制器中实现删除功能。

代码清单28-28　在Controllers文件夹下的AdminController.cs文件中删除用户

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {

    public class AdminController : Controller {
        private UserManager<AppUser> userManager;

        public AdminController(UserManager<AppUser> usr
Mgr) {
            userManager = usrMgr;
        }

        // ...other actions omitted for brevity...

        [HttpPost]
        public async Task<IActionResult> Delete(string
id) {
            AppUser user = await userManager.FindByIdAs
ync(id);
            if (user != null) {
                IdentityResult result = await userManag
er.DeleteAsync(user);
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    AddErrorsFromResult(result);
                }
            } else {
```

```
                ModelState.AddModelError("", "User Not
Found");
            }
            return View("Index", userManager.Users);
        }

        private void AddErrorsFromResult(IdentityResult
 result) {
            foreach (IdentityError error in result.Erro
rs) {
                ModelState.AddModelError("", error.Desc
ription);
            }
        }
    }
}
```

操作方法接收用户的唯一ID作为参数，并使用FindByIdAsync方法找到相应的user对象，以便可以传递给DeleteAsync方法。DeleteAsync方法的返回结果是一个IdentityResult对象。使用前面同样的方式进行处理，以确保向用户显示任何错误。可以通过创建用户，然后单击Index视图中的Delete按钮来测试删除功能，如图28-12所示。

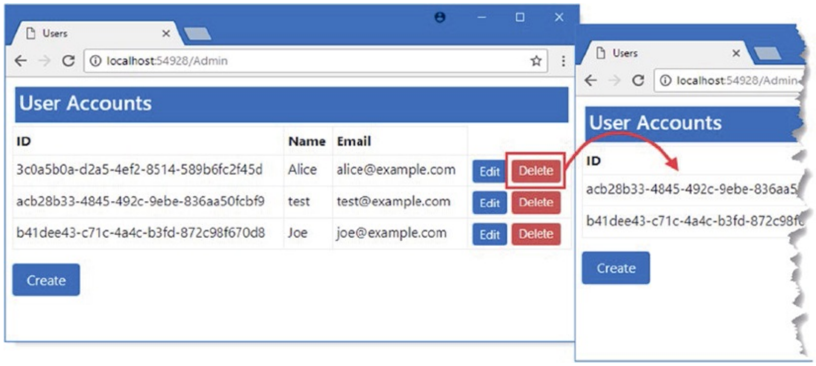


图28-12　删除用户账户

28.4.2　实现编辑功能

要完成管理工具，需要加入对用户账户的电子邮件地址和密码的编辑支持。此刻，它们是用户定义的仅有的两个属性，第30章将介绍如何使用自定义属性来扩展架构。代码清单28-29展示了添加到Admin控制器的Edit操作方法。

代码清单28-29　为Controllers文件夹下的AdminController.cs文件中实现编辑功能

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
```

```
namespace Users.Controllers {

    public class AdminController : Controller {
          private UserManager<AppUser> userManager;
      private IUserValidator<AppUser> userValidator;
      private IPasswordValidator<AppUser> passwordValid
ator;
      private IPasswordHasher<AppUser> passwordHasher;

      public AdminController(UserManager<AppUser> usrMg
r,
              IUserValidator<AppUser> userValid,
              IPasswordValidator<AppUser> passValid,
              IPasswordHasher<AppUser> passwordHash) {
          userManager = usrMgr;
          userValidator = userValid;
          passwordValidator = passValid;
          passwordHasher = passwordHash;
      }
        // ...other action methods omitted for brevity.
..

        public async Task<IActionResult> Edit(string id
) {
            AppUser user = await userManager.FindByIdAs
ync(id);
            if (user != null) {
                return View(user);
            } else {
                return RedirectToAction("Index");
            }
        }

        [HttpPost]
        public async Task<IActionResult> Edit(string id
```

```
, string email,
                string password) {
            AppUser user = await userManager.FindByIdAs
ync(id);
            if (user != null) {
                user.Email = email;
                IdentityResult validEmail
                    = await userValidator.ValidateAsync
(userManager, user);
                if (!validEmail.Succeeded) {
                    AddErrorsFromResult(validEmail);
                }
                IdentityResult validPass = null;
                if (!string.IsNullOrEmpty(password)) {
                    validPass = await passwordValidator
.ValidateAsync(userManager,
                        user, password);
                    if (validPass.Succeeded) {
                        user.PasswordHash = passwordHas
her.HashPassword(user,
                            password);
                    } else {
                        AddErrorsFromResult(validPass);
                    }
                }
                if ((validEmail.Succeeded && validPass
== null)
                        || (validEmail.Succeeded
                        && password != string.Empty &&
validPass.Succeeded)) {
                    IdentityResult result = await userM
anager.UpdateAsync(user);
                    if (result.Succeeded) {
                        return RedirectToAction("Index"
);
                    } else {
```

```
                        AddErrorsFromResult(result);
                    }
                }
            } else {
                ModelState.AddModelError("", "User Not
Found");
            }
            return View(user);
        }
        private void AddErrorsFromResult(IdentityResult
 result) {
            foreach (IdentityError error in result.Erro
rs) {
                ModelState.AddModelError("", error.Desc
ription);
            }
        }
    }
}
```

Edit操作通过在Index视图中嵌入ID字符串的GET请求来调用FindByIdAsync方法以获取表示用户的AppUser对象。

更复杂的实现则接收POST请求，其中带有用户ID、新的电子邮件地址和密码。必须执行几个任务才能完成Edit操作。

第一个任务是验证接收的值。此刻，正在使用简单的user对象，尽管第30章将展示如何自定义为用户存储的数据。即便如此，也仍然需要验证数据以确保不会违反28.3.3节和28.3.4节中的自定义验证策略。要验证电子邮件地址，可以这样做：

```
...
user.Email = email;
IdentityResult validEmail = await userValidator.ValidateAsync(userManager, user);
if (!validEmail.Succeeded) {
    AddErrorsFromResult(validEmail);
}
...
```

为IUserValidator<AppUser>在控制器的构造函数中添加依赖，以便可以验证新的电子邮件地址。注意，在验证之前，必须修改Email属性，因为ValidateAsync方法仅仅接收user对象。

下一步是更改密码（如果已经提供）。ASP.NET Core Identity存储密码的哈希值而不是密

码本身，这是为了防止密码被盗。接下来获取通过验证的密码并生成将存储到数据库中的哈希值，这将在第29章中演示。密码通过实现了IPasswordHasher<AppUser>接口的类被转换为哈希值，可通过声明依赖注入的构造函数参数来获取。IPasswordHasher接口定义了HashPassword方法，该方法接收一个字符串参数并返回对应的哈希值，如下所示：

```
...
if (!string.IsNullOrEmpty(password)) {
    validPass = await passwordValidator.ValidateAsync(userManager, user, password);
    if (validPass.Succeeded) {
        user.PasswordHash = passwordHasher.HashPassword(user, password);
    } else {
        AddErrorsFromResult(validPass);
    }
}
...
```

对AppUser类的所做修改不会被存储，直到UpdateAsync方法被调用，如下所示：

```
...
if ((validEmail.Succeeded && validPass == null) || (validEmail.Succeeded
        && password != string.Empty && validPass.Succeeded)) {
    IdentityResult result = await userManager.UpdateAsync(user);
    if (result.Succeeded) {
        return RedirectToAction("Index");
    } else {
        AddErrorsFromResult(result);
    }
}
...
```

创建视图

最后的组件是用来向用户显示当前值并允许将新值提交给控制器的视图，代码清单28-30展示了创建在Views/Admin文件夹下的Edit.cshtml文件中的内容。

代码清单28-30　Views/Admin文件夹下的Edit.cshtml文件的内容

```
@model AppUser
```

```
<div class="bg-primary m-1 p-1"><h4>Edit User</h4></div
>

<div asp-validation-summary="All" class="text-danger"><
/div>

<form asp-action="Edit" method="post">
    <div class="form-group">
        <label asp-for="Id"></label>
        <input asp-for="Id" class="form-control" disabl
ed />
    </div>
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input name="password" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Save<
/button>
    <a asp-action="Index" class="btn btn-secondary">Can
cel</a>
</form>
```

Edit视图将显示以静态文本显示时无法修改的用户ID，并提供编辑电子邮件地址和密码的表单，如图28-13所示。注意，没有对密码使用标签助手，因为AppUser类不包含密码信息，只有哈希值

保存在数据库中。

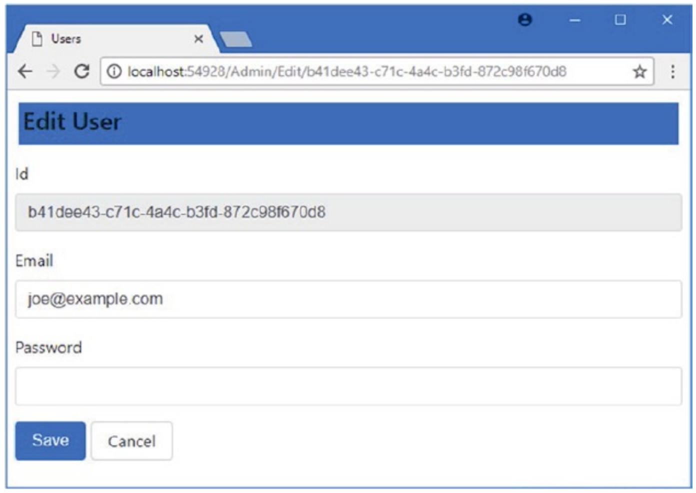


图28-13　编辑用户账户

最后的修改是从Startup类中注释掉用户验证设置，以便用于用户名的默认字符可用，如代码清单28-31所示。由于数据库中的某些账户是在更改验证设置之前创建的，因此无法编辑它们，这是因为用户名不会通过验证。另外，当电子邮件地址被验证时，验证将作用于整个user对象，结果就是用户账户不能修改。

代码清单28-31　在Startup.cs文件中禁用自定义的验证设置

```
...
public void ConfigureServices(IServiceCollection servic
es) {

    services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
    services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();

    services.AddDbContext<AppIdentityDbContext>(options
 =>
        options.UseSqlServer(
            Configuration["Data:SportStoreIdentity:Conn
ectionString"]));

    services.AddIdentity<AppUser, IdentityRole>(opts =>
 {
        opts.User.RequireUniqueEmail = true;
        //opts.User.AllowedUserNameCharacters = "abcdef
ghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
    }).AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
...
```

要测试编辑功能，可运行应用，然后访

问/Admin并单击Edit按钮。修改电子邮件地址或者输入新密码，单击Save按钮以更新数据库并返回/Admin。

28.5　小结

本章展示了如何创建使用ASP.NET Core Identity所需的配置和类，并演示了如何应用它们来创建用户管理工具。下一章将展示如何使用ASP.NET Core Identity执行身份验证和授权。

第29章　应用ASP.NET Core Identity

本章将展示如何为上一章创建的用户账号使用ASP.NET Core Identity进行验证和授权。表29-1列出本章要介绍的操作。

表**29-1**　本章要介绍的操作

操作	方法	代码清单
限制访问操作方法	应用Authorize特性	代码清单29-1
验证用户	创建Account控制器以接收用户凭据，使用UserManager类检查凭据	代码清单29-2～代码清单29-5
创建和管理角色	使用UserManager类	代码清单29-6～代码清单29-10
使用角色授权访问操作方法	将用户账户加入角色，使用Authorize特性指定可以访问操作方法的角色	代码清单29-11～代码清单29-18
		代码清单29-19～代

确认管理账户	使用种子数据库自动创建账户	码清单29-24

29.1　准备示例项目

本章将继续使用你在第28章中创建的项目。请运行应用程序并导航到URL地址/Admin，使用Create按钮确认表29-2中的用户账户已存入数据库中。

表**29-2**　本章需要的用户账户

用户名	电子邮件	密码
Joe	joe@example.com	secret123
Alice	alice@example.com	secret123
Bob	bob@example.com	secret123

完成后，访问/Admin，将会展示一个用户列表，其中包含表29-2中的用户（不必介意创建的其

他用户，只要表29-2中的用户存在即可），如图29-1所示。

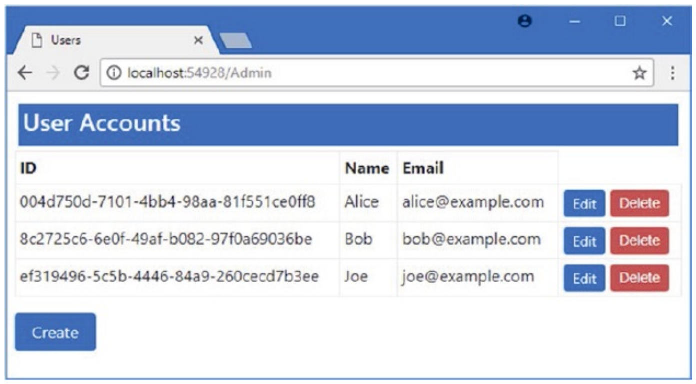


图29-1　执行示例应用

29.2　验证用户

ASP.NET Core Identity最基本的用途就是验证用户。限制访问操作方法的关键工具是Authorize特性，它告诉MVC只有来自已验证用户的请求才应该被处理。代码清单29-1为Home控制器的Index操作方法应用了Authorize特性。

代码清单29-1　在Controllers文件夹下的HomeController.cs文件中限制访问

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

    public class HomeController : Controller {

        [Authorize]
        public ViewResult Index() =>
            View(new Dictionary<string, object> { ["Pla
ceholder"] = "Placeholder" });
    }
}
```

如果启动应用，浏览器将发送请求到默认的URL地址，该URL将针对使用Authorize修饰的操作方法。目前用户无法自行验证，结果如图29-2所示。

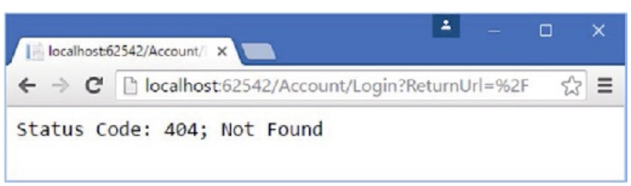


图29-2　访问受保护的action方法的结果

Authorize特性没有指定如何验证用户，也没有

直接链接到ASP.NET Core Identity。验证服务和中间件将跨越整个ASP.NET Core平台，通过修改描述HTTP请求的上下文对象，可以简单流畅地与MVC应用集成，将验证处理的详细结果提供给MVC应用而不用涉及内部细节。

ASP.NET Core平台通过HttpContext对象来提供关于用户的信息，该对象被Authorize特性用来检查当前请求的用户状态和查看用户是否已被验证。HttpContext.User属性返回一个实现了IPrincipla接口的实例，这个接口定义在命名空间System.Security.Principal中。IPrincipal接口中的重要成员包括Identity属性和IsInRole(role)方法。

Identity属性返回接口Iidentity的实现，IIdentity接口描述了请求关联的用户。

如果用户是特定角色的成员，IsInRole(role)方

法则返回true。

你可通过IPrincipal.Identity属性返回IIdentity接口的实现，并通过表29-3中的属性返回一些关于当前用户的信息。

表**29-3**　定义在**IIdentity**接口中的重要属性

属性	描述
AuthenticationType	返回用于用户验证机制的描述字符串
IsAuthenticated	如果用户已经被验证，则返回true
Name	返回当前用户的名字

提　示

第30章将介绍ASP.NET Core Identity中关于IIdentity接口的实现类。

ASP.NET Core Identity中间件使用浏览器返回的Cookie来检查用户是否已经通过验证。如果用户已经通过验证，就把IIdentity.IsAuthenticated属性设置为true。由于当前的示例应用还没有提供验证机制，因此IsAuthenticated属性总是返回false。这将导致验证错误，使客户端被重定向到地址/Account/Login，这是提供验证凭据的默认地址。

浏览器被重定向到地址/Account/Login，但是示例应用没有任何相关的控制器或操作存在，服务器返回404-Not Found响应，导致的错误消息参见图29-2。

改变登录地址

虽然/Account/Login是需要验证时客户端默认的重定向地址，但是在设置ASP.NET Core Identity服务的时候，可以在Startup类的ConfigureServices方法中修改配置选项来指定自己的地址，如下所示：

```
...
services.ConfigureApplicationCookie(opts => opts.LoginPath = "/Users/Login");
...
```

由于不能依赖路由系统来生成URL地址，因此必须使用字面值来设定重定向目标。如果为应用修改了路由模式，就必须确保同时修改了ASP.NET Core Identity设置，以便这个地址还可以到达目标控制器。

29.2.1　准备实现验证

请求虽然以错误结束，但还是展示了ASP.NET Core Identity如何适配标准的ASP.NET请求生命周期。下一步，我们将实现一个接收请求并验证用户的控制器。如代码清单29-2所示，添加一个新的模型类到UserViewModels.cs文件中。

代码清单29-2　在Models文件夹下的UserViewModels.cs文件中添加新的模型类

```
using System.ComponentModel.DataAnnotations;

namespace Users.Models {

    public class CreateModel {
        [Required]
        public string Name { get; set; }
        [Required]
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
    }
    public class LoginModel {
```

```
        [Required]
        [UIHint("email")]
        public string Email { get; set; }

        [Required]
        [UIHint("password")]
        public string Password { get; set; }
    }
}
```

这个新的模型拥有Email和Password属性，它们都使用Required特性进行修饰，所以可以使用模型验证来检查用户是否提供了这些值。还可以使用UIHint特性来修饰这些属性，这将使标签助手在视图中渲染的input元素拥有合适的type属性。

提　示

在真实项目中，客户端验证可以在提交表单到

服务器之前检查用户提供的用户名和密码，详见第27章。

添加名为AccountController.cs的类文件到Controllers文件夹中，定义代码清单29-3所示的控制器。

代码清单29-3　Controllers文件夹下的AccountController.cs文件的内容

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
        public IActionResult Login(string returnUrl) {
            ViewBag.returnUrl = returnUrl;
            return View();
        }
```

```
        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginMod
el details, string returnUrl) {
            return View(details);
        }
    }
}
```

这里还没有实现验证逻辑，下面首先定义视图，然后演示检验用户凭据的过程以及用户登录应用的过程。

虽然还没有验证用户，但Account控制器包含了一些有用的基础架构，可从ASP.NET Core Identity代码中分别说明这些不久就要添加到Login操作方法的内容。

注意，两个版本的Login操作方法都有一个名为returnUrl的参数。当用户访问受限的地址时，他们将被重定向到带有指向受限地址的查询串的地

址/Account/Login，一旦用户被验证，就可以返回这个地址。如果启动应用并访问地址/Home/Index，就可以看到这些。你的浏览器将会被重定向，如下所示：

```
/Account/Login?ReturnUrl=%2FHome%2FIndex
```

ReturnUrl参数的字符串值使得打开页面和安全处理之间的页面重定向处理变得平滑和无缝。

接下来，注意应用于Account控制器的特性。管理用户账号的控制器包含只有验证用户才能使用的功能，例如重置口令。我们最终为控制器应用了Authorize特性，然后为个别操作方法应用了AllowAnonymous特性。这样，默认的操作方法需要验证用户，但是允许未验证用户登录应用。这里还应用了ValidateAntiForgeryToken特性，第24章介绍过，从而与表单元素的标签助手协作以防止跨站

仿冒攻击。

最后的准备步骤是创建用户获取凭据的视图。创建Views/Account文件夹，添加名为Login.cshtml的视图文件，内容如代码清单29-4所示。

代码清单29-4　Views/Account文件夹下的Login.cshtml文件的内容

```
@model LoginModel

<div class="bg-primary m-1 p-1 text-white"><h4>Log In</
h4></div>

<div class="text-danger" asp-validation-summary="All"><
/div>

<form asp-action="Login" method="post">
    <input type="hidden" name="returnUrl" value="@ViewB
ag.returnUrl" />
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control"
/>
    </div>
```

```
    <button class="btn btn-primary" type="submit">Log I
n</button>
</form>
```

Login视图中唯一需要注意的就是hidden输入元素，这里用来保持returnUrl参数。从任何其他方面看，Login都是标准的Razor视图，并且完成了验证的准备工作，演示了未经验证的请求被拦截和重定向的方式。测试这个新的控制器，启动应用，当浏览器请求应用程序的默认地址时，重定向到地址/Account/Login，生成的内容如图29-3所示。

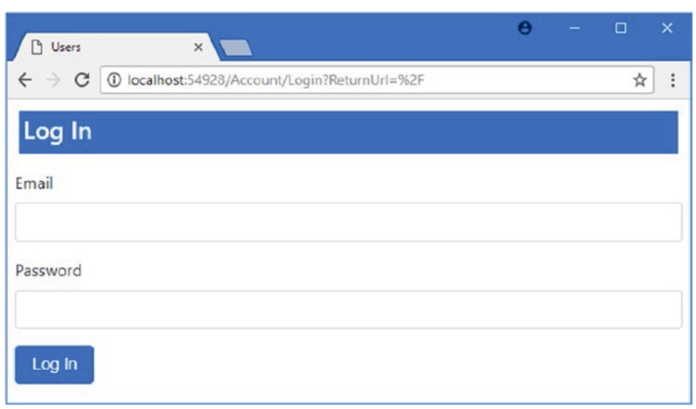


图29-3　生成的内容

29.2.2　添加用户验证

请求保护的操作方法已被正确重定向到Account控制器，但是用户提供的凭据还没有被验证。代码清单29-5完整实现了Login操作方法，并使用ASP.NET Core Identity服务根据存储在数据库中的详细内容来验证用户。

代码清单29-5　在Controllers文件夹下的AccountController.cs文件中添加用户验证

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<AppUser> userManager;
        private SignInManager<AppUser> signInManager;

        public AccountController(UserManager<AppUser> u
serMgr,
                SignInManager<AppUser> signinMgr) {
            userManager = userMgr;
            signInManager = signinMgr;
        }
```

```
        [AllowAnonymous]
        public IActionResult Login(string returnUrl) {
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginMod
el details,
                string returnUrl) {
            if (ModelState.IsValid) {
                AppUser user = await userManager.FindBy
EmailAsync(details.Email);
                if (user != null) {
                    await signInManager.SignOutAsync();
                    Microsoft.AspNetCore.Identity.SignI
nResult result =
                            await signInManager.Passwor
dSignInAsync(
                                user, details.Password,
 false, false);
                    if (result.Succeeded) {
                        return Redirect(returnUrl ?? "/
");
                    }
                }
                ModelState.AddModelError(nameof(LoginMo
del.Email),
                    "Invalid user or password");
            }
            return View(details);
        }
    }
```

```
}
```

以上代码中最简单的部分是获取用于表示用户的AppUser对象，可使用UserManager<AppUser>类的FindByEmailAsync方法来完成：

```
...
AppUser user = await userManager.FindByEmailAsync(details.Email);
...
```

这个方法使用已经创建的电子邮件地址来定位用户。另外一些候选的用于定位用户的方式有使用ID、用户名以及登录等。这里使用电子邮件地址进行登录，因为这种方式可由大多数面向互联网的Web应用采用，而且在企业级应用中变得越来越流行。

如果存在包含用户指定的电子邮件地址的账户，下一步就是执行验证，由于需要使用SignInManager<AppUser>类，因此添加通过依赖注

入获取的构造函数参数。可使用SignInManager类执行两个验证步骤：

```
...
await signInManager.SignOutAsync();
Microsoft.AspNetCore.Identity.SignInResult result =
    await signInManager.PasswordSignInAsync(user, detai
ls.Password, false, false);
...
```

SignOutAsync方法取消用户拥有的任何现有会话，PasswordSignIn方法执行验证。PasswordSignInAsync方法的参数包括用户对象、用户提供的密码以及用于控制身份验证的Cookie是否应该持久的bool参数（已被禁用）。这个bool参数还决定了如果密码正确，账号是否应该被锁定（也已禁用）。

PasswordSignInAsync方法的返回结果是一个SignInResult对象，其中定义了名为Succeeded的bool属性来标识验证是否已经成功。

在这个示例中，检查Succeeded属性。如果值为true，就将用户重定向到returnUrl地址；否则，添加验证错误消息，然后重新显示Login视图以便用户可以重新尝试登录。

作为验证过程的一部分，ASP.NET Core Identity添加了一个Cookie到响应中，以便浏览器在随后的请求中携带这个Cookie，它将用来标识用户的会话和关联的用户账号。不必直接创建或管理这个Cookie，它将被ASP.NET Core Identity中间件自动处理。

考虑双因子验证

执行单因子验证，以便用户可以使用他们已经提前知道的信息——密码进行验证。

ASP.NET Core Identity同样支持双因子验证，此时用户需要一些额外信息，通常是在验证用户身份时给予用户的信息。最常见的是来自SecureID令牌的值以及来自邮件或短信的验证码（严格来说，双因子可以是任何因素，包括指纹、虹膜扫描、声音识别，虽然这些是大多数Web应用很少需要的选项）。

由于攻击者需要知道用户的密码并且需要第二个因子，因此电子邮件账户或手机的安全性被加强了。

有两个原因导致本书不会展示双因子验证。首先，因为需要做大量的准备工作，比如设置分发作为第二个因子的电子邮件地址和短信的基础架构，并且实现验证逻辑，所有这些都超出了本书的讨论范围。

其次，因为双因子验证迫使用户记住要跳到额外的步骤进行验证，比如记住自己的电话号码或者保持SecureID令牌在手边，有些方式不便于在Web应用中使用。

如果对双因子验证感兴趣，建议基于第三方系统（比如Google）进行验证，这将允许用户选择自己希望的双因子验证额外的安全信息。第30章将演示第三方验证。

29.2.3　测试验证

为了测试验证，可启动应用并请求地址/Home/Index。当被重定向到地址/Account/Login后，输入本章开头列出的用户信息（例如，电子邮件地址joe@example.com 和密码secret123）。单击Log In按钮，浏览器将被重定向到地

址/Home/Index。但是，此时将提交验证Cookie，使得被授权访问这个操作方法，如图29-4所示。

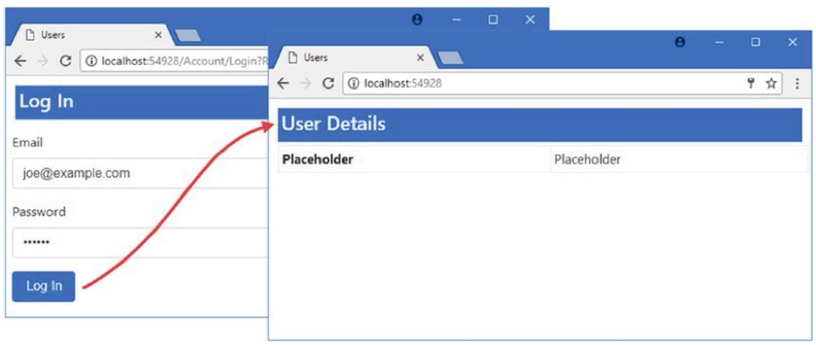


图29-4　验证用户

提　示

使用浏览器的开发人员工具可以查看用于标识验证请求的Cookie。

29.3　使用角色授权用户

之前，特性Authorize以最基本的方式使用，这将允许任何经过验证的用户执行操作方法。基于用户的成员角色，还可以优化授权，对根据哪些用户执行哪些操作进行精细控制。

角色仅仅是在应用中自定义的表示一组活动权限的定制标签。几乎所有的应用都区分可以执行管理功能的用户和没有管理功能的用户。在角色的世界中，可通过创建Administrators角色并赋予用户来实现授权。用户可以隶属于多个角色，角色的权限可以是粗放的，也可以是细粒度的。因此，可以使用单独的角色来区分可以执行基本任务的管理员（比如创建新的账号），以及可以执行更敏感操作的用户（比如访问付款数据）。

ASP.NET Core Identity负责管理定义在应用中

的角色，并跟踪隶属于每个角色的用户成员，但是它不知道每个角色的意义，这些信息包含在应用的MVC部分，并基于角色成员来限制对操作方法的访问。

ASP.NET Core Identity提供了基于强类型的名为RoleManager<T>的基类来访问和管理角色，这里的T就是存储机制中表示角色的类。Entity Framework Core使用名为IdentityRole的类，其中定义的属性如表29-4所示。

表**29-4　IdentityRole**类定义的属性

属性	描述
Id	定义角色的唯一标识
Name	定义角色名称
Users	返回表示角色成员的IdentityUserRole对象集合

如果希望扩展内置的功能，可以创建应用特定的角色类，第30章将说明用户对象，但是这里将继续使用IdentityRole类，这是因为该类可以完成大多数应用需要的所有工作。第28章在配置应用的时候已经告诉ASP.NET Core Identity使用IdentityRole来表示角色，详见Startup类的ConfigureService方法中的代码：

```
...
services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
    //opts.User.AllowedUserNameCharacters = "abcdefghij
klmnopqrstuvwxyz";
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
}).AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

AddIdentity方法的类型参数指定了将用来表示用户和角色的类。在示例应用中，AppUser类用来

表示用户，内置的IdentityRole类用来表示角色。

29.3.1　创建与删除角色

为了演示如何使用角色，下面创建一个管理工具来管理它们，首先创建可以用来创建和删除角色的操作方法。在Controllers文件夹中创建名为RoleAdminController.cs的类文件，用它定义代码清单29-6所示的控制器。

代码清单29-6　Controllers文件夹下的RoleAdminController.cs文件的内容

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore
;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {

    public class RoleAdminController : Controller {
        private RoleManager<IdentityRole> roleManager;
        public RoleAdminController(RoleManager<Identity
Role> roleMgr) {
```

```
            roleManager = roleMgr;
        }

        public ViewResult Index() => View(roleManager.R
oles);

        public IActionResult Create() => View();

        [HttpPost]
        public async Task<IActionResult> Create([Requir
ed]string name) {
            if (ModelState.IsValid) {
                IdentityResult result
                    = await roleManager.CreateAsync(new
 IdentityRole(name));
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    AddErrorsFromResult(result);
                }
            }
            return View(name);
        }

        [HttpPost]
        public async Task<IActionResult> Delete(string
id) {
            IdentityRole role = await roleManager.FindB
yIdAsync(id);
            if (role != null) {
                IdentityResult result = await roleManag
er.DeleteAsync(role);
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    AddErrorsFromResult(result);
```

```
                }
            } else {
                ModelState.AddModelError("", "No role f
ound");
            }
            return View("Index", roleManager.Roles);
        }

        private void AddErrorsFromResult(IdentityResult
 result) {
            foreach (IdentityError error in result.Erro
rs) {
                ModelState.AddModelError("", error.Desc
ription);
            }
        }
    }
}
```

角色使用RoleManager<T>类进行管理，这里的T是用来表示角色的类（示例应用使用了内置的IdentityRole类）。RoleAdminController构造函数定义了基于RoleManager<IdentityRole>的构造函数依赖，当控制器被创建时，可通过依赖注入获取。

RoleManager<T>类定义了表29-5所示的方法和属性，它们允许创建和管理角色。

表**29-5　RoleManager<T>**类定义的成员

成员	描述
CreateAsync(role)	创建新角色
DeleteAsync(role)	删除特定角色
FindByIdAsync(id)	使用ID发现角色
FindByNameAsync(name)	使用角色名称发现角色
RoleExistsAsync(name)	如果特定名称的角色存在，则返回true
UpdateAsync(role)	为特定角色存储更新
Roles	返回定义角色的枚举

新控制器的Index操作方法将显示应用中的所有角色。Create操作方法被用来显示和接收表单，来自表单的信息将被CreateAsync方法用来创建新的角色。Delete操作方法接收POST类型的请求和唯一的角色标识，使用FindByIdAsync方法来定位ID表

示的角色对象，并使用DeleteAsync方法从应用中删除角色。

1．创建视图

为了在应用中显示角色的详情，创建Views/RoleAdmin文件夹，在其中添加名为Index.cshtml的视图文件，内容如代码清单29-7所示。

代码清单29-7　Views/RoleAdmin文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<IdentityRole>

<div class="bg-primary m-1 p-1"><h4>Roles</h4></div>

<div class="text-danger" asp-validation-summary="ModelO
nly"></div>

<table class="table table-sm table-bordered table-borde
red">
    <tr><th>ID</th><th>Name</th><th>Users</th><th></th>
</tr>
    @if (Model.Count() == 0) {
        <tr><td colspan="4" class="text-center">No Role
```

```
s</td></tr>
    } else {
        foreach (var role in Model) {
            <tr>
                <td>@role.Id</td>
                <td>@role.Name</td>
                <td identity-role="@role.Id"></td>
                <td>
                    <form asp-action="Delete" asp-route
-id="@role.Id" method="post">
                        <a class="btn btn-sm btn-primar
y" asp-action="Edit"
                           asp-route-id="@role.Id">Edit
</a>
                        <button type="submit"
                                class="btn btn-sm btn-d
anger">
                            Delete
                        </button>
                    </form>
                </td>
            </tr>
        }
    }
</table>
<a class="btn btn-primary" asp-action="Create">Create</
a>
```

Index视图使用表格来显示应用角色的详细信息，其中的第三列使用了定制的元素属性，如下所示：

```
...
<td identity-role="@role.Id"></td>
...
```

为了显示每个角色相关的成员列表，需要在视图中包含大量的代码。为了保持视图简单，添加一个名为RoleUsersTagHelper.cs的类文件到Ingrastructure文件夹中，用它定义代码清单29-8所示的标签助手。

代码清单29-8　Infrastructure文件夹下的RoleUsersTagHelper.cs文件的内容

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore
;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Users.Models;

namespace Users.Infrastructure {

    [HtmlTargetElement("td", Attributes = "identity-rol
e")]
    public class RoleUsersTagHelper : TagHelper {
        private UserManager<AppUser> userManager;
```

```
        private RoleManager<IdentityRole> roleManager;

        public RoleUsersTagHelper(UserManager<AppUser>
usermgr,
                                  RoleManager<IdentityR
ole> rolemgr) {
            userManager = usermgr;
            roleManager = rolemgr;
        }

        [HtmlAttributeName("identity-role")]
        public string Role { get; set; }

        public override async Task ProcessAsync(TagHelp
erContext context,
                TagHelperOutput output) {

            List<string> names = new List<string>();
            IdentityRole role = await roleManager.FindB
yIdAsync(Role);
            if (role != null) {
                foreach (var user in userManager.Users)
 {
                    if (user != null
                        && await userManager.IsInRoleAs
ync(user, role.Name)) {
                        names.Add(user.UserName);
                    }
                }
            }

            output.Content.SetContent(names.Count == 0
?
                "No Users" : string.Join(", ", names));
        }
    }
```

```
}
```

这个标签助手处理td元素的identity-role属性，用它接收被处理角色的ID。RoleManager<IdentityRole>和UserNamager<AppUser>对象允许查询Identity数据库来构建角色中的用户名列表。代码清单29-9将这个标签助手添加到视图的导入文件中，并且添加@using expression，以便在视图中不通过命名空间使用Entity Framework Core中的类型。

代码清单29-9　在Views文件夹下的ViewImports.cshtml文件中添加标签助手

```
@using Users.Models
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper Users.Infrastructure.*, Users
```

然后，添加名为Create.cshtml的视图文件到Views/RoleAdmin文件夹中，并且添加代码清单29-

10所示的标记来支持添加角色。

代码清单29-10　Views/RoleAdmin文件夹下的Create.cshtml文件的内容

```
@model string

<div class="bg-primary m-1 p-1"><h4>Create Role</h4></d
iv>

<div asp-validation-summary="All" class="text-danger"><
/div>

<form asp-action="Create" method="post">
    <div class="form-group">
        <label for="name"></label>
        <input name="name" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Creat
e</button>
    <a asp-action="Index" class="btn btn-secondary">Can
cel</a>
</form>
```

创建角色时唯一需要的表单数据是角色名称，这就是使用string类型作为Create视图的视图模型类的原因。借用模型验证的优势可以确认在提交表单的时候用户提供了值，但是不值得为了这么简单的

任务创建专门的模型类。相反，观察代码清单29-6中接收POST请求的Create操作方法，你会看到直接在参数中使用了Required验证特性。这与在模型类中使用这个特性有同样的效果，而且允许获得内置的模型验证带来的好处。

2．测试、创建与删除角色

为了测试新的控制器，启动应用后导航到URL地址/RoleAdmin，单击Create按钮，在input元素中输入名字，然后单击另一个Create按钮。新的角色将保存到数据库中，在浏览器被重定向到Index操作后将显示出来，如图29-5所示。单击Delete按钮，从应用中删除这个角色。

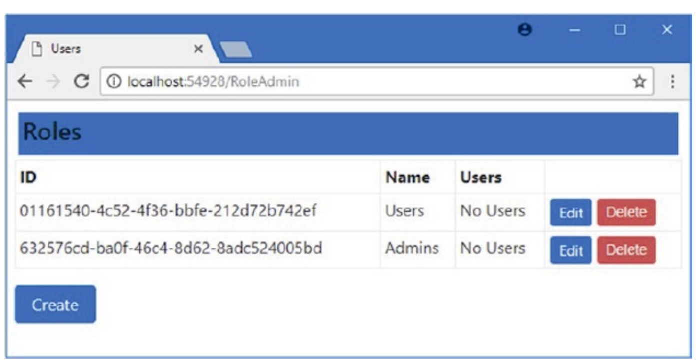


图29-5　创建新角色

29.3.2　管理角色成员

下一步是在角色中添加和删除用户。这个过程并不复杂，但是它调用了来自RoleManager类的角色数据，并将它们与单个用户的详情相关联。

首先定义一些视图模型类，它们用来表示角色的成员资格，并从用户那里接收一组新的成员资格，代码清单29-11展示了在Models文件夹中创建的UserViewModels.cs文件。

代码清单29-11　添加视图模型到UserViewModels.cs文件中

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using Microsoft.AspNetCore.Identity;

namespace Users.Models {
    public class CreateModel {
        [Required]
        public string Name { get; set; }
        [Required]
```

```
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
    }

    public class LoginModel {
        [Required]
        [UIHint("email")]
        public string Email { get; set; }
        [Required]
        [UIHint("password")]
        public string Password { get; set; }
    }

    public class RoleEditModel {
        public IdentityRole Role { get; set; }
        public IEnumerable<AppUser> Members { get; set;
 }
        public IEnumerable<AppUser> NonMembers { get; s
et; }
    }

    public class RoleModificationModel {
        [Required]
        public string RoleName { get; set; }
        public string RoleId { get; set; }
        public string[] IdsToAdd { get; set; }
        public string[] IdsToDelete { get; set; }
    }
}
```

RoleEditModel类表示系统中角色及用户的详细信息，根据用户是否是角色的成员进行分类。

RoleModification- Model类表示针对角色的一组更新。

代码清单29-12展示了RoleAdmin控制器中新添加的操作方法，它们使用代码清单29-11所示的视图模型来管理角色成员。

代码清单29-12　在Controllers文件夹下的RoleAdminController.cs文件中添加操作方法

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;

namespace Users.Controllers {

    public class RoleAdminController : Controller {
        private RoleManager<IdentityRole> roleManager;
        private UserManager<AppUser> userManager;
        public RoleAdminController(RoleManager<Identity
Role> roleMgr,
                                            UserManager<AppUser>
 userMrg) {
            roleManager = roleMgr;
            userManager = userMrg;
```

```
        }

        // ...other action methods omitted for brevity.
..

        public async Task<IActionResult> Edit(string id
) {

            IdentityRole role = await roleManager.FindB
yIdAsync(id);
            List<AppUser> members = new List<AppUser>();

            List<AppUser> nonMembers = new List<AppUser
>();
            foreach (AppUser user in userManager.Users)
 {
                var list = await userManager.IsInRoleAs
ync(user, role.Name)
                    ? members : nonMembers;
                list.Add(user);
            }
            return View(new RoleEditModel {
                Role = role,
                Members = members,
                NonMembers = nonMembers
            });
        }

        [HttpPost]
        public async Task<IActionResult> Edit(RoleModif
icationModel model) {
            IdentityResult result;
            if (ModelState.IsValid) {
                foreach (string userId in model.IdsToAd
d ?? new string[] { }) {
                    AppUser user = await userManager.Fi
```

```
ndByIdAsync(userId);
                    if (user != null) {
                        result = await userManager.AddT
oRoleAsync(user,
                            model.RoleName);
                        if (!result.Succeeded) {
                            AddErrorsFromResult(result);

                        }
                    }
                }
                foreach (string userId in model.IdsToDe
lete ?? new string[] { }) {
                    AppUser user = await userManager.Fi
ndByIdAsync(userId);
                    if (user != null) {
                        result = await userManager.Remo
veFromRoleAsync(user,
                            model.RoleName);
                        if (!result.Succeeded) {
                            AddErrorsFromResult(result);

                        }
                    }
                }
            }
            if (ModelState.IsValid) {
                return RedirectToAction(nameof(Index));
            } else {
                return await Edit(model.RoleId);
            }
        }

        private void AddErrorsFromResult(IdentityResult
 result) {
```

```
            foreach (IdentityError error in result.Erro
rs) {
                ModelState.AddModelError("", error.Desc
ription);
            }
        }
    }
}
```

在GET版本的Edit操作方法中，大部分代码用来生成针对所选角色的成员集合和非成员集合。一旦所有用户分类完成，一个新的RoleEditModel实例就被传递给View方法，以便这些数据可以使用默认视图显示出来。

POST版本的Edit操作方法负责在角色中添加或删除用户。UserManager<T>类提供了一些处理角色的方法，如表29-6所示。

表**29-6　UserManager<T>**类定义的角色相关方法

方法	描述
AddToRoleAsync(user,name)	将用户ID添加到具有指定名称的角色中

GetRoleAsync(user)	返回用户的角色成员名称列表
IsInRoleAsync(user,name)	如果用户是指定角色的成员，返回true
RemoveFromRoleAsync(user, name)	从指定名称的角色中删除用户

这些方法的奇怪之处在于角色相关的操作方法基于角色名称来执行操作，即便角色拥有唯一的标识。正因为如此，RoleModificationModel视图模型类有一个RoleName属性。

创建Edit.cshtml文件并把它添加到Views/RoleAdmin文件夹中，用它定义代码清单29-13所示的标记以允许用户编辑角色成员。

代码清单29-13　Views/RoleAdmin文件夹下的Edit.cshtml文件的内容

```
@model RoleEditModel

<div class="bg-primary m-1 p-1 text-white"><h4>Edit Role</h4></div>
```

```
<div asp-validation-summary="All" class="text-danger"><
/div>

<form asp-action="Edit" method="post">
    <input type="hidden" name="roleName" value="@Model.
Role.Name" />
    <input type="hidden" name="roleId" value="@Model.Ro
le.Id" />

    <h6 class="bg-info p-1 text-white">Add To @Model.Ro
le.Name</h6>
    <table class="table table-bordered table-sm">
        @if (Model.NonMembers.Count() == 0) {
            <tr><td colspan="2">All Users Are Members</
td></tr>
        } else {
            @foreach (AppUser user in Model.NonMembers)
 {
                <tr>
                    <td>@user.UserName</td>
                    <td>
                        <input type="checkbox" name="Id
sToAdd" value="@user.Id">
                    </td>
                </tr>
            }
        }
    </table>

    <h6 class="bg-info p-1 text-white">Remove From @Mod
el.Role.Name</h6>
    <table class="table table-bordered table-sm">
        @if (Model.Members.Count() == 0) {
            <tr><td colspan="2">No Users Are Members</t
d></tr>
```

```
        } else {
            @foreach (AppUser user in Model.Members) {
                <tr>
                    <td>@user.UserName</td>
                    <td>
                        <input type="checkbox" name="Id
sToDelete" value="@user.Id">
                    </td>
                </tr>
            }
        }
    </table>
    <button type="submit" class="btn btn-primary">Save<
/button>
    <a asp-action="Index" class="btn btn-secondary">Can
cel</a>
</form>
```

Edit视图中包含两个表格：一个用于不是所选角色成员的用户，另一个用于是所选角色成员的用户。每个用户名后跟着允许修改成员状态的复选框。两个表格被包含在将发送到Edit操作方法的表单中，数据将被绑定到RoleModificationModel类，从而可以轻松地访问角色成员的变更列表。

测试和编辑角色成员

为了测试角色成员，启动应用并导航到URL地址/RoleAdmin。如果需要，创建名为Users的新角色。单击Edit按钮，你将看到应用中的用户显示为角色的非成员用户，如图29-6所示。

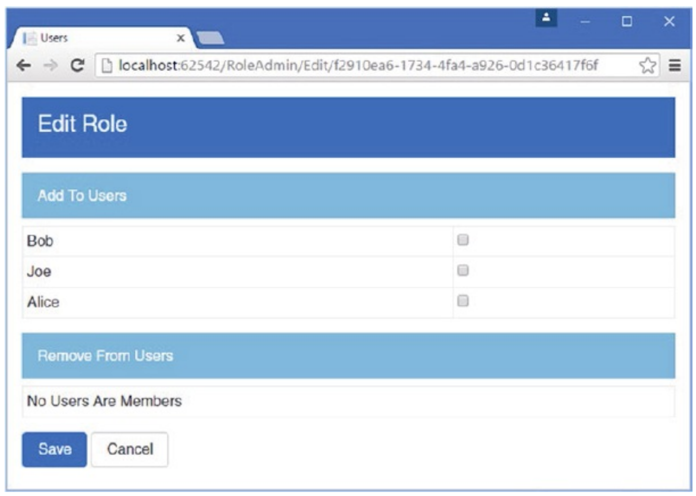


图29-6　显示和编辑角色成员

选中复选框以添加Alice和Joe（在本章开始的时候，这两个账号已添加到ASP.NET Core Identity系统中），然后单击Save按钮。在角色列表中，你将看到Alice和Joe现在位于角色成员列表中，如图29-7所示。

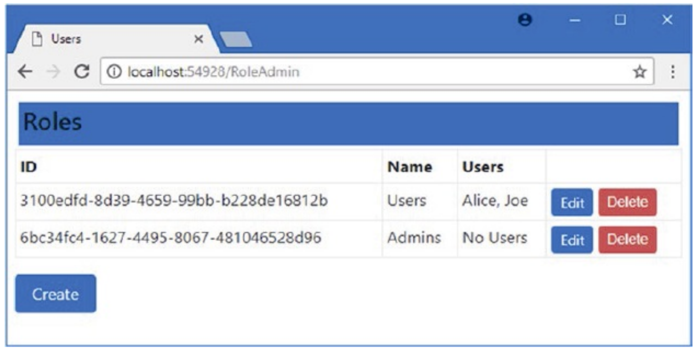


图29-7　管理角色成员

29.3.3　使用角色进行授权

示例应用现在有了角色，它们可以通过Authorize特性作为授权的基础。为了易于测试基于角色的授权，在Account控制器中添加Logout操作方法，如代码清单29-14所示，这将允许用户退出后以另一个用户的身份重新登录并查看角色成员的效果。

代码清单29-14　在Controllers文件夹下的AccountController.cs文件中添加Logout操作方法

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
```

```
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<AppUser> userManager;
        private SignInManager<AppUser> signInManager;

        // ...other action methods omitted for brevity.
..

        [Authorize]
        public async Task<IActionResult> Logout() {
            await signInManager.SignOutAsync();
            return RedirectToAction("Index", "Home");
        }
    }
}
```

下一步是更新Home控制器以加入新的操作方法，并将认证用户的信息传递给视图，如代码清单29-15所示。

代码清单29-15　扩展Controllers文件夹下的HomeController.cs文件

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
```

```
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

    public class HomeController : Controller {

        [Authorize]
        public IActionResult Index() => View(GetData(na
meof(Index)));

        [Authorize(Roles = "Users")]
        public IActionResult OtherAction() => View("Ind
ex",
            GetData(nameof(OtherAction)));
        private Dictionary<string, object> GetData(stri
ng actionName) =>
            new Dictionary<string, object> {
                ["Action"] = actionName,
                ["User"] = HttpContext.User.Identity.Na
me,
                ["Authenticated"] = HttpContext.User.Id
entity.IsAuthenticated,
                ["Auth Type"] = HttpContext.User.Identi
ty.AuthenticationType,
                ["In Users Role"] = HttpContext.User.Is
InRole("Users")
            };
    }
}
```

Index操作方法的Authorize特性没有变化，但是在为OtherAction操作方法应用这个特性时设置了

Roles属性，指定只有Users角色的成员可以访问。这里还定义了GetData方法，以返回关于用户标识的一些基本信息，可使用HttpContext对象的属性获取这些信息。

提　示

Authorize特性也可以基于独立用户名的列表授权访问。对于小项目来说，这是一个很有吸引力的特性，但是这意味着每次要授权的用户发生变化时，都不得不修改控制器中的代码，也这意味着不得不再次完成测试和发布周期。使用角色来授权可以将应用从独立账户的变化中隔离出来，并且允许通过存储在ASP.NET Core Identity中的成员来控制

应用的访问。

最后，修改Views/Home文件夹下的Index.cshtml文件，该文件由Home控制器的两个操作方法使用。添加目标为Account控制器的Logout操作的链接，如代码清单29-16所示。

代码清单29-16　在Views/Home文件夹下的Index.cshtml文件中添加退出链接

```
@model Dictionary<string, object>

<div class="bg-primary m-1 p-1 text-white"><h4>User Details</h4></div>

<table class="table table-sm table-bordered m-1 p-1">
    @foreach (var kvp in Model) {
        <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
    }
</table>

@if (User?.Identity?.IsAuthenticated ?? false) {
    <a asp-controller="Account" asp-action="Logout"
       class="btn btn-danger">Logout</a>
}
```

为了测试验证，启动应用并导航到/Home/Index，浏览器将被重定向以便输入用户凭据。由于应用于Index操作的Authorize特性允许任何通过验证的用户访问，因此从表29-2中选择哪个用户进行验证显得并不重要。

但是，如果访问URL地址/Home/OtherAction，你在表29-2中所做的选择就会导致不同，因为只有Alice和Joe是Users角色的成员，这对访问OtherAction操作是必需的。如果以Bob用户身份登录，浏览器将被重定向到/Account/AccessDenied，这适用于当用户不能访问操作方法时。为了处理这种情况，这里在 Account控制器中添加AccessDenied操作，以便有一个操作可以处理这种请求，如代码清单29-17所示。

提　示

设置 IdentityOptions.Cookies.ApplicationCookie.AccessDen
属性，改变/Account /AccessDenied地址。

代码清单29-17　在AccountController.cs文件中添加操作方法

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<AppUser> userManager;
        private SignInManager<AppUser> signInManager;

        public AccountController(UserManager<AppUser> u
serMgr,
```

```
                SignInManager<AppUser> signinMgr) {
            userManager = userMgr;
            signInManager = signinMgr;
        }

        // ...other action methods omitted for brevity.
..

        [AllowAnonymous]
        public IActionResult AccessDenied() {
            return View();
        }
    }
}
```

要为AccessDenied操作提供视图，可在Views/Account文件夹中创建名为AccessDenied.cshtml的文件，然后添加代码清单29-18所示的内容。

代码清单29-18　Views/Account文件夹下的AccessDenied.cshtml文件的内容

```
<div class="bg-danger mb-1 p-2 text-white"><h4>Access D
enied</h4></div>
<a asp-action="Index" asp-controller="Home" class="btn
btn-primary">OK</a>
```

启动应用并访问/Account/Login，以bob@example.com身份登录。当验证过程完成后，浏览器将被重定向到/Home/Index，这将显示账户的详请，如图29-8左边的截图所示，这表明Bob不是Users角色的成员。现在访问/Home/OtherAction，Bob不是所要求角色的成员，浏览器被重定向到URL地址/Account/AccessDenied，如图29-8右边的截图所示。

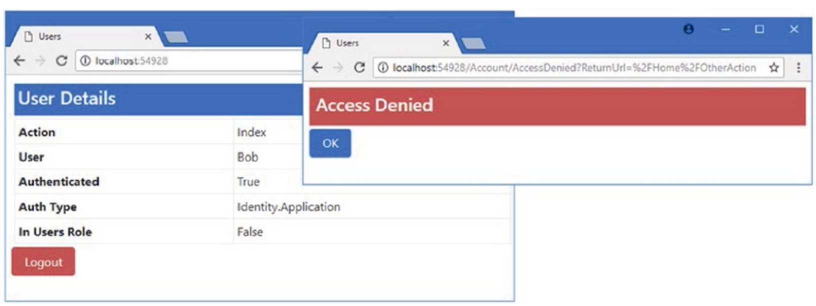


图29-8　使用基于角色的验证

提　示

角色在用户登录的时候加载，这意味着如果改变当前已登录用户的角色，那么在注销和重新登录之前，更改不会生效。

29.4　播种数据库

在示例项目中，访问Admin和RoleAdmin控制器是不受限制的。需要创建用户和角色，但Admin和RoleAdmin控制器是用户管理工具，如果使用Authorize特性保护了它们，就不能授权任何凭据来访问它们了，特别是在第一次部署应用之前。

解决方案是当应用启动的时候，使用一些初始数据来播种数据库。如代码清单29-19所示，添加一些新的配置数据到appsettings.json文件中以指定

将要创建的账户的详情。

代码清单29-19　添加配置数据到appsettings.json文件中

```
{
  "Data": {
    "AdminUser": {
      "Name": "Admin",
      "Email": "admin@example.com",
      "Password": "secret",
      "Role": "Admins"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocal
DB;Database=IdentityUsers;Trusted_Con
nection=True;MultipleActiveResultSets=true"
    }
  }
}
```

Data: AdminUser类别提供了创建账户需要的4个值，并赋予了用于管理工具的角色。

警　告

将口令放在明文配置文件中意味着当部署应用并第一次初始化数据时，必须将修改默认口令作为部署过程的一部分。

然后，在AppIdentityDbContext类中添加一个静态方法，如代码清单29-20所示。创建默认账户的代码不需要定义在这个类中，但是作者把它用在了这里的项目中。

代码清单29-20　在Models文件夹下的AppIdentityDbContext.cs文件中添加静态方法

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore
;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System;
```

```
using System.Threading.Tasks;

namespace Users.Models {

    public class AppIdentityDbContext : IdentityDbConte
xt<AppUser> {

        public AppIdentityDbContext(DbContextOptions<Ap
pIdentityDbContext> options)
            : base(options) { }

        public static async Task CreateAdminAccount(ISe
rviceProvider serviceProvider,
            IConfiguration configuration) {

            UserManager<AppUser> userManager =
                serviceProvider.GetRequiredService<User
Manager<AppUser>>();
            RoleManager<IdentityRole> roleManager =
                serviceProvider.GetRequiredService<Role
Manager<IdentityRole>>();

            string username = configuration["Data:Admin
User:Name"];
            string email = configuration["Data:AdminUse
r:Email"];
            string password = configuration["Data:Admin
User:Password"];
            string role = configuration["Data:AdminUser
:Role"];
            if (await userManager.FindByNameAsync(usern
ame) == null) {
                if (await roleManager.FindByNameAsync(r
ole) == null) {
                    await roleManager.CreateAsync(new I
dentityRole(role));
```

```
                }

                AppUser user = new AppUser {
                    UserName = username,
                    Email = email
                };

                IdentityResult result = await userManag
er
                    .CreateAsync(user, password);
                if (result.Succeeded) {
                    await userManager.AddToRoleAsync(us
er, role);
                }
            }
        }
    }
}
```

CreateAdminAccount方法接收一个IServiceProvider对象，它用来获取UserManager、RoleManager和IConfiguration对象，还可以用来从appsetting.json文件中获取数据。CreateAdminAccount方法中的代码会检查用户是否已经存在，如果不存在，则创建用户并赋予指定的角色。如果角色也不存在，就创建角色。代码清单

29-21在Startup类中添加了一些代码，可在设置和配置应用之后调用CreateAdminAccount方法。

代码清单29-21　在Users文件夹下的Startup.cs文件中调用数据库方法

```
...
public void Configure(IApplicationBuilder app) {
    app.UseStatusCodePages();
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseAuthentication();
    app.UseMvcWithDefaultRoute();
    AppIdentityDbContext.CreateAdminAccount(app.Applica
tionServices,
        Configuration).Wait();
}
...
```

由于正在通过IApplicationBuilder.ApplicationServices提供程序来访问受限的服务，因此还必须在Program类中禁用依赖注入范围验证特性，如代码清单29-22所示。

代码清单29-22　在Users文件夹下的Program.cs文件中禁用依

赖注入范围验证特性

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Users {
    public class Program {
        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] ar
gs) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .UseDefaultServiceProvider(options =>
                    options.ValidateScopes = false)
                .Build();
    }
}
```

现在，Identity数据库中有了可信的默认账号，可以使用Authorize特性来保护Admin和RoleAdmin控制器，代码清单29-23展示了对Admin控制器所做

的修改。

代码清单29-23　在Users文件夹下的AdminController.cs文件中限制访问

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

    [Authorize(Roles = "Admins")]
    public class AdminController : Controller {

        // ...statements omitted for brevity...
    }
}
```

代码清单29-24展示了在RoleAdmin控制器中所做的修改。

代码清单29-24　在Controllers文件夹下的RoleAdminController.cs文件中限制访问

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {
    [Authorize(Roles = "Admins")]
    public class RoleAdminController : Controller {
        // ...statements omitted for brevity...
    }
}
```

启动应用并访问URL地址/Admin或/RoleAdmin。如果已经以其他用户身份登录，则必须先注销。否则，系统将提醒你输入凭据，可以使用账号admin@example.com和密码secret进行验证以访问管理功能。

29.5 小结

本章展示了如何使用ASP.NET Core Identity进行验证和授权，说明了如何收集和验证用户凭据，以及如何基于用户所属的角色来限制对操作方法的

访问。下一章将演示ASP.NET Core Identity提供的高级特性。

第30章　ASP.NET Core Identity进阶

本章将通过展示ASP.NET Core Identity提供的一些高级特性来完成对ASP.NET Core Identity的介绍，演示如何通过为AppUser类上定义自定义属性来扩展数据库架构，以及如何在不删除ASP.NET Core Identity数据库中数据的情况下使用数据库迁移来应用这些属性。此外，本章还将说明ASP.NET Core Identity如何支持声明（claim）的概念，并展示如何通过策略灵活地授权对操作方法的访问。最后，本章通过展示ASP.NET Core Identity如何使用第三方对用户进行身份验证来结束本章。可使用Google账户来演示身份验证，但ASP.NET Core Identity还内置对微软、Facebook和Twitter账户的支持。表30-1列出了本章要介绍的操作。

表30-1　本章要介绍的操作

操作	方法	代码清单
存储用户自定义数据	添加属性到AppUser类，并更新Identity数据库	代码清单30-1～代码清单30-3
执行精细授权	使用声明	代码清单30-4～代码清单30-7
创建自定义声明	使用声明转换	代码清单30-8和代码清单30-9
使用声明评估用户访问	创建策略	代码清单30-10～代码清单30-14
使用策略访问资源	评估操作方法中的策略	代码清单30-15～代码清单30-20
允许使用第三方执行验证	接收来自验证提供器的声明，比如微软、Google和Facebook	代码清单30-21～代码清单30-24

30.1　准备示例项目

本章将继续使用你在第28章创建且在第29章扩

展的Users项目。启动应用并确认数据库中已经有一些用户。图30-1展示了数据库的状态，其中包含来自上一章的用户Admin、Alice、Bob和Joe。为了检查这些用户，启动应用并访问/Admin，以 Admin 用户身份进行验证，使用电子邮件地址admin@example.com 和密码secret。

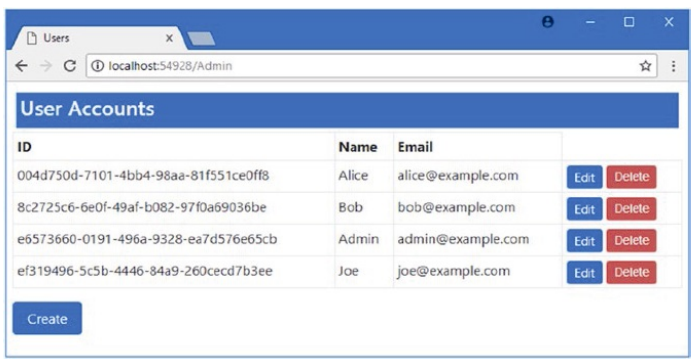


图30-1　Identity数据库中的初始用户

本章还需要一些角色。导航到/RoleAdmin，然后创建名为Users和Employees的角色并为这些角色分配用户，如表30-2所示。

表**30-2**　示例应用要求的角色和成员

角色	成员
Users	Alice、Joe
Employees	Alice、Bob

图30-2展示了RoleAdmin控制器的必要角色配置。

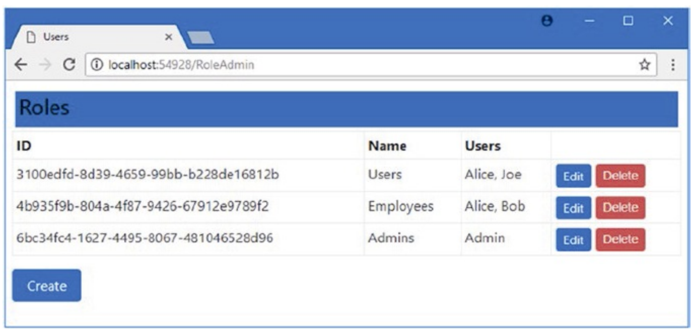


图30-2　配置本章要求的角色

30.2　添加自定义用户属性

在第28章，当创建AppUser类来表示用户的时候，基类定义了用户的基本属性，比如电子邮件地址和电话号码。

大多数应用程序需要保存更多的用户信息，包括持久化的应用程序首选项和详细信息，比如地址。简而言之，任何运行应用程序有用且在会话之间保留的数据。由于ASP.NET Core Identity系统默认使用Entity Framework Core来保存数据，因此定义额外的用户信息意味着为AppUser类添加属性并使用Entity Framework Core创建存储它们所需的数据库模式。

代码清单30-1展示了如何为AppUser类添加两个简单属性以表示用户生活的城市和资格等级。

代码清单30-1　在Models文件夹下的AppUser.cs文件中添加属性

```
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

    public enum Cities {
        None, London, Paris, Chicago
    }

```

```
    public enum QualificationLevels {
        None, Basic, Advanced
    }

    public class AppUser : IdentityUser {
        public Cities City { get; set; }
        public QualificationLevels Qualifications { get
; set; }
    }
}
```

枚举Cities和QualificationLevels定义了用于城市的值以及资格水平的不同级别。这些枚举由添加到AppUser类的City和Qualification属性使用。

在代码清单30-2中，添加到Home控制器的操作方法允许用户查看和编辑它们的City和Qualification属性。

代码清单30-2　在Controllers文件夹下的HomeController.cs文件中添加对自定义用户属性的支持

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Users.Models;

```

```
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace Users.Controllers {
    public class HomeController : Controller {
        private UserManager<AppUser> userManager;

        public HomeController(UserManager<AppUser> user
Mgr) {
            userManager = userMgr;
        }

        [Authorize]
        public IActionResult Index() => View(GetData(na
meof(Index)));

        [Authorize(Roles = "Users")]
        public IActionResult OtherAction() => View("Ind
ex",
            GetData(nameof(OtherAction)));

        private Dictionary<string, object> GetData(stri
ng actionName) =>
            new Dictionary<string, object> {
                ["Action"] = actionName,
                ["User"] = HttpContext.User.Identity.Na
me,
                ["Authenticated"] = HttpContext.User.Id
entity.IsAuthenticated,
                ["Auth Type"] = HttpContext.User.Identi
ty.AuthenticationType,
                ["In Users Role"] = HttpContext.User.Is
InRole("Users"),
                ["City"] = CurrentUser.Result.City,
                ["Qualification"] = CurrentUser.Result.
```

```
Qualifications
            };

        [Authorize]
        public async Task<IActionResult> UserProps() {
            return View(await CurrentUser);
        }

        [Authorize]
        [HttpPost]
        public async Task<IActionResult> UserProps(
                [Required]Cities city,
                [Required]QualificationLevels qualifica
tions) {
            if (ModelState.IsValid) {
                AppUser user = await CurrentUser;
                user.City = city;
                user.Qualifications = qualifications;
                await userManager.UpdateAsync(user);
                return RedirectToAction("Index");
            }
            return View(await CurrentUser);
        }

        private Task<AppUser> CurrentUser =>
            userManager.FindByNameAsync(HttpContext.Use
r.Identity.Name);
    }
}
```

新的CurrentUser属性使用UserManager<AppUser>类来获取表示当前用户的

AppUser实例。AppUser实例可作为GET版本的UserProps操作方法的视图模型，POST方法则使用它来更新City和QualificationLevel属性的值。

已经更新的GetData方法则返回包含当前用户自定义属性的值的字典，这意味着这些属性的值将在Index和OtherAdction操作方法的视图中展示出来。

为了给UserProps操作方法提供视图，在Views/Home文件夹中添加名为UserProps.cshtml的文件，并添加代码清单30-3所示的标记。

代码清单30-3　Views/Home文件夹下的UserProps.cshtml文件的内容

```
@model AppUser

<div class="bg-primary m-1 p-1 text-white"><h4>@Model.UserName</h4></div>

<div asp-validation-summary="All" class="text-danger"><
```

```
/div>

<form asp-action="UserProps" method="post">
    <div class="form-group">
        <label asp-for="City"></label>
        <select asp-for="City" class="form-control"
                asp-items="@new SelectList(Enum.GetName
s(typeof(Cities)))">
            <option disabled selected value="">Select a
 City</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Qualifications"></label>
        <select asp-for="Qualifications" class="form-co
ntrol"
            asp-items="@new SelectList(Enum.GetNames(ty
peof(QualificationLevels)))">
                <option disabled selected value="">Sele
ct a City</option>
        </select>
    </div>

    <button type="submit" class="btn btn-primary">Submi
t</button>
    <a asp-action="Index" class="btn btn-secondary">Can
cel</a>
</form>
```

UserProps视图包含一个具有select元素的表单，其中填充了代码清单30-1中的枚举值。当表单被提交时，从ASP.NET Core Identity中获取表示当

前用户的AppUser实例，用户的自定义属性的值被更新为当前选中的值，如下所示：

```
...
AppUser user = await CurrentUser;
user.City = city;
user.Qualifications = qualifications;
await userManager.UpdateAsync(user);
return RedirectToAction("Index");
...
```

注意，必须通过调用UpdateAsync方法显式地告诉用户管理器为用户更新数据库记录以反映更新。以前不必这样做是因为在更新ASP.NET Core Identity的方法中已经调用了UpdateAsync方法，但是当直接修改属性时，你有责任通知用户管理器执行更新。

30.2.1 准备数据库迁移

用以支持新增属性的所有应用内部支持已经就绪，剩下的就是更新数据库，以便其中的数据表可

以存储自定义属性的值。

Entity Framework Core没有对处理种子数据提供集成支持，在创建迁移以禁用种子的时候必须小心，如代码清单30-4所示；否则，代码清单30-1中新添加到模型类的属性将会导致错误。种子语句可以在创建和应用数据库迁移之后再次启用。

代码清单30-4　在Users文件夹下的Startup.cs文件禁用数据库播种

```
...
public void Configure(IApplicationBuilder app) {
    app.UseStatusCodePages();
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseAuthentication();
    app.UseMvcWithDefaultRoute();
    //    AppIdentityDbContext.CreateAdminAccount(app.ApplicationServices,
    //        Configuration).Wait();
}
...
```

在禁用种子语句的情况下，下一步是创建新的

数据库迁移文件，其中包含用来更新数据库架构的SQL命令。使用命令行窗口或PowerShell窗口在Users项目文件夹中执行如下命令：

```
dotnet ef migrations add CustomProperties
```

当命令完成的时候，你将会在Migrations文件夹中发现一个名称中含有CustomProperties的新文件。确切的名称中还包含数字标识，但是如果打开这个文件，你将会发现一个包含名为Up的方法的C#类，Up方法执行为支持向数据库添加自定义属性所需的SQL命令。另外一个名为Down的方法用于执行将数据库降级到上一个模式的命令。

下一步是迁移数据库到新的模式，可通过执行以下命令完成：

```
dotnet ef database update
```

当命令执行完之后，数据库中存储用户数据的表将包含新的表示自定义属性的列。

警　告

在包含真实用户数据的产品数据库上执行数据库迁移时，务必小心。创建包含删除列或整个表的迁移非常容易，这可能导致毁灭性的影响。确保彻底测试数据库迁移的影响，确保备份关键数据以防万一。

30.2.2　测试自定义属性

要测试数据库迁移的影响，可启动应用程序，并使用已标识的用户之一进行验证（例如，使用电子邮件地址alice@example.com和密码secret123）。一旦验证通过，你将会看到City和QualificationLevel属性的默认值。这两个属性可以通过访问/Home/UserProps进行修改，选择新的值，单击Submit按钮，这将会更新数据库并重定向到/Home，这里将显示新的值，如图30-3所示。

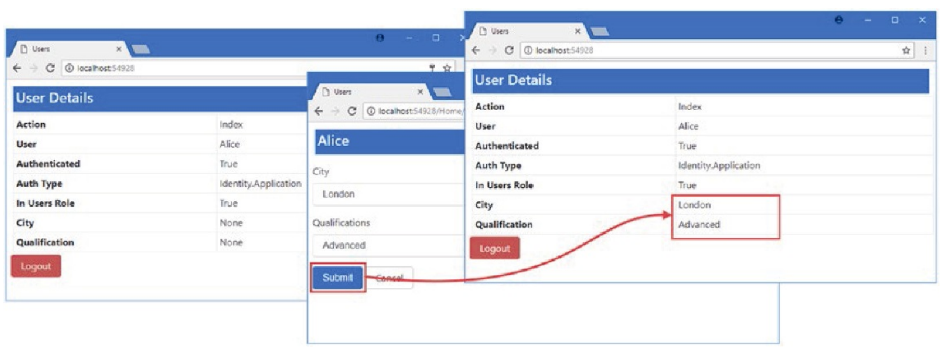


图30-3　使用自定义用户属性

30.3　使用声明和策略

较早的用户管理系统（比如ASP.NET

Membership）是ASP.NET Core Identity的前身，应用程序本身假定自己是所有用户信息的权威来源，基本上把应用看作封闭的世界，只信任应用自身的数据。

这是软件开发中很久远的处理方式。在第29章中，当基于保存在数据库中的凭据验证用户，然后基于这些凭据授权访问时，你已经看到了一些示例。在本章中，当添加属性到AppUser类的时候，可以做同样的事情。管理用户的验证和授权所需的任何信息都来自应用。对于许多Web应用来说，这是一种完美的方式，这也是本章如此深入地演示这些技术的原因。

ASP.NET Core Identity还支持另外一种管理用户的替代方式——使用声明，当MVC应用程序并不是用户信息的唯一来源时同样工作良好，相对于传统方式，以更加灵活、流畅的方式授权用户。

提　示

并一定需要使用声明，如第29章所述，ASP.NET Core Identity完美提供了验证和授权服务，完全不需要理解声明。

30.3.1　声明

声明是关于用户的一段信息，它还包括关于信息来源的信息。理解声明的最简单方式是进行一些实际的演示，否则任何讨论都太过抽象而无用。为了开始演示，在Controllers文件夹中添加名为ClaimsController.cs的类文件，并用它定义代码清单

30-5所示的控制器。

提　示

你可能对这个示例中的代码和类感到一点迷惑。现在不必担心细节，只要坚持下去，直至看到操作方法和定义的视图的输出。重要的是，这将有助于理解声明。

代码清单30-5　Controllers文件夹下的ClaimsController.cs文件的内容

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {
```

```
    public class ClaimsController : Controller {

        [Authorize]
        public ViewResult Index() => View(User?.Claims)
;
    }
}
```

可以通过多种途径来获取用户相关的声明。User属性（也可以通过HttpContext.User属性）返回一个ClaimsPrincipal对象，这也是这个示例中使用的方式。关联到用户的声明集合可通过ClaimsPrincipal类访问，该类的重要成员如表30-3所示。

表**30-3　ClaimsPrincipal**类的重要成员

成员	描述
Identity	获取关联到当前用户的IIdentity
FindAll(type) FindAll(<predicate>)	返回特定类型或匹配条件谓词的声明

FindFirst(type) FindFirst(<predicate>)	返回第一个特定类型或匹配条件谓词的声明
HasClaim(type,value) HasClaim(<predicate>)	如果用户拥有特定声明类型的特定值，或者拥有匹配条件谓词的声明，就返回true
IsInRole(name)	如果用户是特定角色的成员，就返回true

如第28章所述，HttpContext.User.Identity属性将返回一个实现了IIdentity接口的对象，在ASP.NET Core Identity中是ClaimsIdentity对象。表30-4展示了定义在ClaimsIdentity类中的重要成员。

表**30-4**　定义在**ClaimsIdentity**类中的重要成员

成员	描述
Claims	返回表示用户声明的Claim对象的枚举
AddClaim(claim)	为用户标识添加声明
AddClaims(claims)	为用户标识添加一组声明
HasClaim(predicate)	如果用户标识包含匹配特定谓词的声明，就返回true

RemoveClaim(claim)	从用户标识中删除声明

还有其他的方法和属性可用，但是表30-4中的这些经常用于Web应用，原因显而易见，它们演示了声明（Claim）如何适用于广泛的ASP.NET Core平台。

代码清单30-5使用Controller.User属性来获取ClaimsPrincipal对象并传递Claims属性值作为视图模型给默认视图。Cliam对象表示关于用户的单片数据，Claim类定义的属性如表30-5所示。

表**30-5　Claim**类定义的属性

属性	描述
Issuer	返回提供声明的系统的名称
Subject	返回声明所表示用户的ClaimsIdentity对象
Type	返回声明所表示的信息类型

Value	返回声明所表示的信息片段

为了显示用户所关联声明的详情，创建Views/Claims文件夹，在其中创建名为Index.cshtml的文件，并添加代码清单30-6所示的标记内容。

代码清单30-6　Views/Claims文件夹下的Index.cshtml文件的内容

```
@model IEnumerable<System.Security.Claims.Claim>

<div class="bg-primary m-1 p-1 text-white"><h4>Claims</
h4></div>

<table class="table table-sm table-bordered">
    <tr>
        <th>Subject</th><th>Issuer</th><th>Type</th><th
>Value</th>
    </tr>
    @if (Model == null || Model.Count() == 0) {
        <tr><td colspan="4" class="text-center">No Clai
ms</td></tr>
    } else {
        @foreach (var claim in Model.OrderBy(x => x.Typ
e)) {
            <tr>
                <td>@claim.Subject.Name</td>
                <td>@claim.Issuer</td>
                <td identity-claim-type="@claim.Type"><
```

```
/td>
                <td>@claim.Value</td>
            </tr>
        }
    }
</table>
```

Index视图使用表格来显示视图模型中提供的每个声明。Claim.Type属性是用于微软架构的URI，不是特别有用。常用的架构可作为System.Security.Claims.ClaimTypes的字段值，所以为了使Index视图的输出更易读，可添加自定义属性到td元素以显示Type属性的值：

```
...
<td identity-claim-type="@claim.Type"></td>
...
```

添加名为ClaimTypeTagHelper.cs的类文件到Infrastructure文件夹中，并用它创建标记助手以转换属性的值到更易读的字符串中，如代码清单30-7所示。

代码清单30-7　Infrastructure文件夹下的ClaimTypeTagHelper.cs文件的内容

```
using System.Linq;
using System.Reflection;
using System.Security.Claims;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Users.Infrastructure {

    [HtmlTargetElement("td", Attributes = "identity-cla
im-type")]
    public class ClaimTypeTagHelper : TagHelper {
        [HtmlAttributeName("identity-claim-type")]
        public string ClaimType { get; set; }

        public override void Process(TagHelperContext c
ontext,
                                     TagHelperOutput ou
tput) {
            bool foundType = false;
            FieldInfo[] fields = typeof(ClaimTypes).Get
Fields();
            foreach (FieldInfo field in fields) {
                if (field.GetValue(null).ToString() ==
ClaimType) {
                    output.Content.SetContent(field.Nam
e);
                    foundType = true;
                }
            }
            if (!foundType) {
                output.Content.SetContent(ClaimType.Spl
it('/', '.').Last());
```

```
                }
            }
        }
    }
```

要了解为什么只创建使用声明的控制器而不解释声明是什么，可启动应用程序，以用户Alice的身份进行验证（使用电子邮件地址alice@example.com以及密码secret123）。一旦通过验证，就访问/Claims来查看用户关联的声明，如图30-4所示。

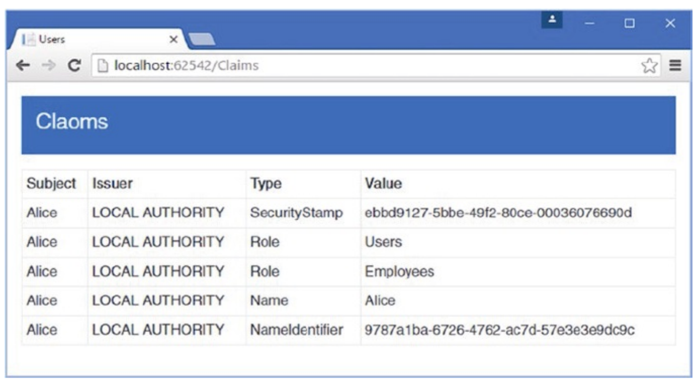


图30-4　Claims控制器的Index操作的输出

通过图30-4你很难弄清楚细节，所以这里重写了表30-6所示的内容。

表**30-6**　图**30-4**中展示的内容

Subject	Issuer	Type	Value
Alice	LOCAL AUTHORITY	SecurityStamp	唯一ID
Alice	LOCAL AUTHORITY	Role	Users
Alice	LOCAL AUTHORITY	Role	Employees
Alice	LOCAL AUTHORITY	Name	Alice
Alice	LOCAL AUTHORITY	NameIdentifier	Alice的用户ID

表30-6展示了声明的一些重要方面，第29章在实现传统的身份验证与授权功能时，已经使用过它们。可以看到某些声明涉及用户标识（Name声明是Alice，NameIdentifier声明是Alice在ASP.NET Core Identity数据库中的唯一用户标识）。其他声明展示了角色成员，表30-6中有两个角色声明，反映了Alice被赋予Users和Employees角色。

当这些信息被表示为一组声明时，区别在于可

以确定数据的来源。表30-6中所有声明的Issuer属性都被设置为LOCAL AUTHORITY，这表示用户的身份由应用程序建立。

现在你已经看到一些示例声明，可以更容易地描述声明了：声明是任何关于用户的可用于应用程序的信息，包括用户的标识和角色成员资格，并且如你所见，在早期章节中定义的关于用户的信息由ASP.NET Core Identity自动提供。虽然一开始声明令人困惑，但是如同MVC应用程序的其他方面一样，一旦看清它们是如何工作的，它们就变得不那么困难了。

30.3.2　创建声明

有意思的是，可以从多种来源获取声明，而不仅仅依赖本地数据库中的用户信息。30.4节会演示实际的示例，但是此刻将通过添加一个类到示例项

目中来模拟提供声明信息的系统。代码清单30-8展示了添加到Infrastructure文件夹的LocationClaimsProvider.cs文件的内容。

代码清单30-8　Infrastructure文件夹下的LocationClaimsProvider.cs文件的内容

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;

namespace Users.Infrastructure {

    public class LocationClaimsProvider : IClaimsTransf
ormation {

        public Task<ClaimsPrincipal> TransformAsync(Cla
imsPrincipal principal) {
            if (principal != null && !principal.HasClai
m(c =>
                    c.Type == ClaimTypes.PostalCode)) {
                ClaimsIdentity identity = principal.Ide
ntity as ClaimsIdentity;
                if (identity != null && identity.IsAuth
enticated
                        && identity.Name != null) {
                    if (identity.Name.ToLower() == "ali
ce") {
                        identity.AddClaims(new Claim[]
{
```

```
                            CreateClaim(ClaimTypes.Post
alCode, "DC 20500"),
                            CreateClaim(ClaimTypes.Stat
eOrProvince, "DC")
                        });
                    } else {
                        identity.AddClaims(new Claim[]
{
                            CreateClaim(ClaimTypes.Post
alCode, "NY 10036"),
                            CreateClaim(ClaimTypes.Stat
eOrProvince, "NY")
                        });
                    }
                }
            }
            return Task.FromResult(principal);
        }

        private static Claim CreateClaim(string type, s
tring value) =>
            new Claim(type, value, ClaimValueTypes.Stri
ng, "RemoteClaims");
    }
}
```

由接口IClaimsTransformation定义的TransformAsync方法接收ClaimsPrincipal对象并检查有效性，然后将Identity属性的值转型为ClaimsIdentity对象。Name属性的值用于创建关于

用户邮编和州的声明。

这个类能够模拟中心化的HR数据库系统，作为员工位置信息的权威来源。例如，为注册声明的来源，可在Startup类的ConfigureServices方法中定义来源，如代码清单30-9所示。

代码清单30-9　在Users文件夹下的Startup.cs文件中启用声明转换

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;

namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }
```

```
        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddTransient<IPasswordValidator<Ap
pUser>,
                CustomPasswordValidator>();
            services.AddTransient<IUserValidator<AppUse
r>,
                CustomUserValidator>();
            services.AddSingleton<IClaimsTransformation,

                LocationClaimsProvider>();

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
(opts => {
                opts.User.RequireUniqueEmail = true;
                //opts.User.AllowedUserNameCharacters =
 "abcdefghijklmnopqrstuvwxyz";
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
                .AddDefaultTokenProviders();

            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

每当一个请求被接收时，声明转换中间件就调用LocationClaimsProvider.AddClaims方法，模拟HR数据源并创建自定义声明。可以通过启动应用程序，以用户身份验证，然后访问/Claim来查看自定义声明的效果。图30-5展示了Alice的声明。你可能需要先注销再重新登录才能查看变化。

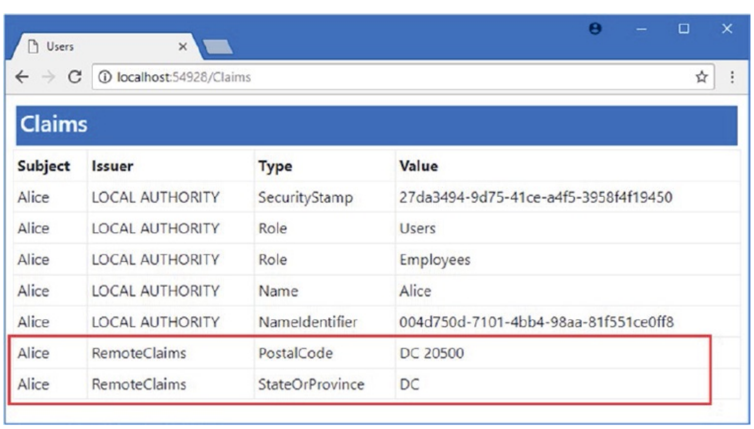


图30-5　为用户定义额外的声明

从多个位置获取声明意味着应用程序不必重复在其他地方已经持有的数据，并允许集成外部数据。Claim.Issuer属性指出声明来自何方，这有助于判断数据的准确程度，以及在应用程序中赋予的权重。从中心化的HR数据库中获取的位置信息可能比从外部邮件列表供应商获取的数据更准确和可靠。

创建自定义的标识声明

如果希望向应用程序添加自定义的本地声明，可以在创建新用户时执行此操作。UserManager<T>类提供了可用于定义本地声明的AddClaimAsync和AddClaimsAsync方法，随后声明会存储在数据库中，并在用户被验证时自动提取（这意味着不需要

依赖与声明转换特性）。但是，在使用这些方法之前，请考虑如何将数据存储在最新状态，以及如何从数据源中动态检索数据以更好地服务于应用程序。声明用于授权检查，而陈旧的声明数据允许用户访问本应该被禁止访问的应用程序区域，并阻止已经被授权访问的应用程序区域。

30.3.3 使用策略

一旦一些声明可用，就可以用相比标准角色更为灵活的方式来管理用户对应用程序的访问。角色的问题在于它们是静态的，一旦用户被分配到角色，用户就保持成员直到角色被显式移除。例如，大公司中的长期雇员最终获得内部系统难以置信的访问权限的原因，就在于他们被赋予每个新职位所需的角色，但是旧的角色很少被删除。

声明用于构建授权策略，它们是应用程序配置的一部分，并将Authorize特性应用于操作方法或控制器。代码清单30-10展示的简单策略只允许拥有特定声明类型和值的用户访问。

代码清单30-10　在Users文件夹下的Startup.cs文件中创建声明策略

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;

namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {
```

```
            services.AddTransient<IPasswordValidator<Ap
pUser>,
                CustomPasswordValidator>();
            services.AddTransient<IUserValidator<AppUse
r>,
                CustomUserValidator>();
            services.AddSingleton<IClaimsTransformation
, LocationClaimsProvider>();

            services.AddAuthorization(opts => {
                opts.AddPolicy("DCUsers", policy => {
                    policy.RequireRole("Users");
                    policy.RequireClaim(ClaimTypes.Stat
eOrProvince, "DC");
                });
            });

            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));
            services.AddIdentity<AppUser, IdentityRole>
(opts => {
                opts.User.RequireUniqueEmail = true;
                //opts.User.AllowedUserNameCharacters =
 "abcdefghijklmnopqrstuvwxyz";
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
```

```
                .AddDefaultTokenProviders();

            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

AddAuthorization方法用于设置授权策略并提供AuthorizationOptions对象，AuthorizationOption类定义的成员如表30-7所示。

表**30-7　AuthorizationOptions**类定义的成员

成员	描述
DefaultPolicy	返回默认的授权策略，用于在没有使用任何参数的情况下使用Authorize特性时。默认情况下，检查用户是否已经被验证
AddPolicy(name,expression)	用于定义新策略

策略是使用AddPolicy方法定义的，可在AuthorizationPolicyBuilder对象上使用表30-8描述的方法逐步构建策略。

表**30-8　AuthorizationPolicyBuilder**类定义的成员方法

方法	描述
RequireAuthenticatedUser()	要求请求关联于已验证的用户
RequireUserName(name)	要求请求关联于特定用户
RequireClaim(type)	要求用户拥有特定类型的声明。只要求特定类型的声明存在，声明的任何值都可以接收
RequireClaim(type,values)	要求用户拥有特定类型的声明，且拥有特定范围内的某个值。值可以表示为使用逗号分隔的参数，或是IEnumerable<string >类型值
RequireRole(roles)	要求用户是角色的成员。多个角色可以使用逗号分隔的参数或IEnumerable<string >表示，任何角色的成员都可以满足要求
AddRequirements(requirement)	用于添加自定义的需求到策略中

代码清单30-10中的策略要求用户拥有Users角色以及带DC值的StateOrProvince声明。当有多个要求时，必须满足所有的要求才能授权。

AddPolicy方法的第一个参数是应用策略时引用的策略名称。在代码清单30-10中，策略的名称是DCUsers。在代码清单30-11中，这个名称由Home控制器中的Authorize特性使用以应用策略。

代码清单30-11　在Controllers文件夹下的HomeController.cs文件中应用授权策略

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Users.Models;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace Users.Controllers {

    public class HomeController : Controller {
        private UserManager<AppUser> userManager;

        public HomeController(UserManager<AppUser> user
```

```
Mgr) {
            userManager = userMgr;
        }

        [Authorize]
        public IActionResult Index() => View(GetData(na
meof(Index)));
        //[Authorize(Roles = "Users")]
        [Authorize(Policy = "DCUsers")]
        public IActionResult OtherAction() => View("Ind
ex",
            GetData(nameof(OtherAction)));

        // ...other methods omitted for brevity...

        private Task<AppUser> CurrentUser =>
            userManager.FindByNameAsync(HttpContext.Use
r.Identity.Name);
    }
}
```

Policy属性用于指定将用于保护操作方法的策略名称。结果就是当请求针对OtherAction 方法时，将对用户拥有的角色和声明进行组合检查。只有在Alice拥有角色成员和声明的正确组合时，才可以通过运行应用程序进行检查，以不同的身份进行验证，并访问/Home/OtherAction。

创建自定义的策略需求

内置的策略需求会检查特定的值，这是很好的起点，但是并不能处理所有的授权场景。如果某个声明值的访问应当被禁止，那么事情就会变得棘手，内置的策略需求不是为应对这种检查而建立的。

幸运的是，可以通过自定义需求来扩展策略系统，它们是实现了接口IAuthorizationRequirement并自定义授权处理的类，也是评估给定请求的需求的AuthorizationHandler类的子类。为了演示，在Infrastructure文件夹中添加名为BlockUsersRequirement.cs的类文件，如代码清单30-12所示，用它定义自定义需求和处理程序。

代码清单30-12　Infrastructure文件夹下的BlockUsersRequirement.cs文件的内容

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Infrastructure {

    public class BlockUsersRequirement : IAuthorization
Requirement {

        public BlockUsersRequirement(params string[] us
ers) {
            BlockedUsers = users;
        }

        public string[] BlockedUsers { get; set; }
    }

    public class BlockUsersHandler : AuthorizationHandl
er<BlockUsersRequirement> {

        protected override Task HandleRequirementAsync(
                AuthorizationHandlerContext context,
                BlockUsersRequirement requirement) {
            if (context.User.Identity != null && contex
t.User.Identity.Name != null
                && !requirement.BlockedUsers
                    .Any(user => user.Equals(context.Us
er.Identity.Name,
                        StringComparison.OrdinalIgnoreC
ase))) {
                context.Succeed(requirement);
            } else {
                context.Fail();
            }
```

```
            return Task.CompletedTask;
        }
    }
}
```

BlockUserRequirement类是需求指定用来创建策略的数据，在这个示例中是不会被授权访问的用户列表。BlockUsersHandler类的职责是使用需求数据评估授权中的请求，该类派生自AuthorizationHandler<T>类，T是需求的类型。

当授权系统需要检查对资源的访问时，处理程序中的Handle方法被调用。这个方法的参数是AuthorizationHandlerContext对象，其中的重要成员如表30-9所示。

表**30-9　AuthorizationHandlerContext**对象中的重要成员

成员	描述
User	返回请求关联的ClaimsPrincipal对象

Succeed(requirement)	如果请求符合需求，这个方法将被调用，参数对象IAuthorizationRequirement由Handle方法接收
Fail()	如果请求不符合需求，这个方法将被调用
Resource	返回用于授权访问单个应用程序资源的对象

代码清单30-12中的需求处理程序检查用户的名称是否在BlockUsersRequirement对象提供的被禁止的名单中，并分别调用Succeed或Fail方法。应用自定义的授权需求需要执行两处配置变更，如代码清单30-13所示。

代码清单30-13　在Users文件夹下的Startup.cs文件中应用自定义的授权需求

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;
using Microsoft.AspNetCore.Authorization;
```

```
namespace Users {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollectio
n services) {

            services.AddTransient<IPasswordValidator<Ap
pUser>,
                CustomPasswordValidator>();
            services.AddTransient<IUserValidator<AppUse
r>,
                CustomUserValidator>();
            services.AddSingleton<IClaimsTransformation
, LocationClaimsProvider>();
            services.AddTransient<IAuthorizationHandler
, BlockUsersHandler>();

            services.AddAuthorization(opts => {
                opts.AddPolicy("DCUsers", policy => {
                    policy.RequireRole("Users");
                    policy.RequireClaim(ClaimTypes.Stat
eOrProvince, "DC");
                });
                opts.AddPolicy("NotBob", policy => {
                    policy.RequireAuthenticatedUser();
                    policy.AddRequirements(new BlockUse
rsRequirement("Bob"));
                });
            });
```

```
            services.AddDbContext<AppIdentityDbContext>
(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdent
ity:ConnectionString"]));

            services.AddIdentity<AppUser, IdentityRole>
(opts => {
                opts.User.RequireUniqueEmail = true;
                //opts.User.AllowedUserNameCharacters =
 "abcdefghijklmnopqrstuvwxyz";
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric =
false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
            }).AddEntityFrameworkStores<AppIdentityDbCo
ntext>()
                .AddDefaultTokenProviders();

            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
{
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

第一步是将处理程序类与服务提供者注册为IAuthorizationHandler接口的实现。第二步是使用AddRequirements方法将自定义的需求添加到策略中，如下所示：

```
...
opts.AddPolicy("NotBob", policy => {
    policy.RequireAuthenticatedUser();
    policy.AddRequirements(new BlockUsersRequirement("Bob"));
});
...
```

得到的结果是一个策略，它要求用户不是Bob，并且可以使用Authorize特性通过指定策略名称来应用，如代码清单30-14所示。

代码清单30-14　在HomeController.cs文件中应用自定义策略

```
...
//[Authorize(Roles = "Users")]
[Authorize(Policy = "DCUsers")]
public IActionResult OtherAction() => View("Index", GetData(nameof(OtherAction)));

[Authorize(Policy = "NotBob")]
```

```
public IActionResult NotBob() => View("Index", GetData(
nameof(NotBob)));
...
```

如果以Bob身份进行验证，将不能访问URL地址/Home/NotBob，但是其他的账户将被授权访问。

30.3.4　使用策略对资源授权访问

策略也可以对单个资源进行授权访问，资源是应用程序中任意数据项的广义术语，并且相比操作方法级别需要更精细的管理。作为演示，将一个名为ProtectedDocument.cs的类文件添加到Models文件夹中，用它定义一个类，以表示具有所有权属性的文档，如代码清单30-15所示。

代码清单30-15　Models文件夹下的ProtectedDocument.cs文件的内容

```
namespace Users.Models {

    public class ProtectedDocument {
        public string Title { get; set; }
```

```
        public string Author { get; set; }
        public string Editor { get; set; }
    }
}
```

对于真实文档来说，这只是一个占位符，关键是每个文档只能由两种人修改，分别是作者和编辑。真实的文档需要内容以及变更跟踪和许多其他的特性，但是对于这个示例已经足够了。在Controllers文件夹中添加名为DocumentController.cs的类文件，并用它定义代码清单30-16所示的控制器。

代码清单30-16　Controllers文件夹下的DocumentController.cs文件的内容

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Users.Models;

namespace Users.Controllers {

    [Authorize]
    public class DocumentController : Controller {
```

```
        private ProtectedDocument[] docs = new Protecte
dDocument[] {
            new ProtectedDocument { Title = "Q3 Budget"
, Author = "Alice",
                Editor = "Joe"},
            new ProtectedDocument { Title = "Project Pl
an", Author = "Bob",
                Editor = "Alice"}
        };

        public ViewResult Index() => View(docs);

        public ViewResult Edit(string title) {
            return View("Index", docs.FirstOrDefault(d
=> d.Title == title));
        }
    }
}
```

Document控制器维护一个固定的ProtectedDocument对象集合，ProtectedDocument对象用在Index操作方法中，并传递所有的文档给View方法。Edit操作方法基于title参数选择文档。这两个操作方法都使用名为Index.cshtml的视图文件，将它添加到新创建的名为Views/Document的文件夹中，内容如代码清单30-17所示。

代码清单30-17　Views/Document文件夹下的Index.cshtml文件的内容

```
@if (Model is IEnumerable<ProtectedDocument>) {
    <div class="bg-primary m-1 p-1 text-white">
        <h4>Documents (@User?.Identity?.Name)</h4>
    </div>
    <table class="table table-sm table-bordered">
        <tr><th>Title</th><th>Author</th><th>Editor</th
><th></th></tr>
        @foreach (var doc in Model) {
            <tr>
                <td>@doc.Title</td>
                <td>@doc.Author</td>
                <td>@doc.Editor</td>
                <td>
                    <a class="btn btn-sm btn-primary" a
sp-action="Edit"
                       asp-route-title="@doc.Title">
                        Edit
                    </a>
                </td>
            </tr>
        }
    </table>
} else {
    <div class="bg-primary m-1 p-1">
        <h4>Editing @Model.Title (@User?.Identity?.Name
)</h4>
    </div>
    <div class="m-1 p-1">
        Document editing feature would go here...
    </div>
    <a asp-action="Index" class="btn btn-primary">Done<
```

```
/a>
}
<a asp-action="Logout" asp-controller="Account" class="
btn btn-danger">Logout</a>
```

如果视图模型是ProtectedDocument对象序列，那么Index视图将以表格的形式，在每行中显示一个文档、作者和编辑的姓名以及用于链接到Edit操作的链接。如果视图模型是单个ProtectedDocument对象，那么Index视图将显示一些占位符，用于为应用程序提供编辑功能。

此刻，只有应用于DocumentController类的Authorize特性的授权限制，这意味着任何用户都可以编辑任意文档，而不仅仅是作者和编辑。可以通过执行应用程序来查看，访问/Document，以任意应用程序用户进行验证，然后单击文档的Edit按钮。例如，图30-6表明用户Joe可以编辑Project Plan文档。

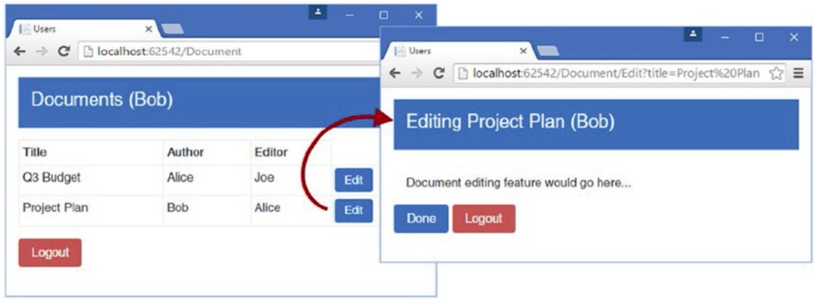


图30-6　编辑文档

创建资源授权策略和处理程序

很难在操作方法级别限制对单个文档的访问，因为Authorize特性是在操作方法执行前评估的。这意味着在提取和检查ProtectedDocument对象之前，哪些用户应该被允许访问文档的决定已经做完。

这个问题的解决方案是创建授权策略和处理程序，它们知道如何处理ProtectedDocument对象，并在用户的详情被展现之后在操作方法中使用它们。为了演示，在Infrastructure文件夹中添加名为DocumentAuthorization.cs的类文件，并用它定义代码清单30-18所示的类。

代码清单30-18　Infrastructure文件夹夹下的DocumentAuthorization.cs文件的内容

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Users.Models;

namespace Users.Infrastructure {

    public class DocumentAuthorizationRequirement : IAu
thorizationRequirement {
        public bool AllowAuthors { get; set; }
        public bool AllowEditors { get; set; }
    }

    public class DocumentAuthorizationHandler
        : AuthorizationHandler<DocumentAuthorizationReq
uirement> {

        protected override Task HandleRequirementAsync(
                AuthorizationHandlerContext context,
                DocumentAuthorizationRequirement requir
ement) {
            ProtectedDocument doc = context.Resource as
 ProtectedDocument;
            string user = context.User.Identity.Name;
            StringComparison compare = StringComparison
.OrdinalIgnoreCase;
            if (doc != null && user != null &&
                (requirement.AllowAuthors && doc.Author
.Equals(user, compare))
                || (requirement.AllowEditors && doc.Edi
tor.Equals(user, compare))) {
```

```
                context.Succeed(requirement);
            } else {
                context.Fail();
            }
            return Task.CompletedTask;
        }
    }
}
```

AuthorizationHandlerContext对象提供了Resource属性，从而提供对可以检查授权的对象的访问。DocumentAuthorizationHandler类检查Resource属性是否为ProtectedDocument对象，如果是，就检查当前用户是否为作者和编辑，以及DocumentAuthorizationRequirement对象是否允许编辑或作者访问文档。

代码清单30-19已经注册DocumentAuthorizationHandler类作为DocumentAuthorizationRequirement需求的处理类，并定义了具有这一需求的策略。

代码清单30-19　在Users文件夹下的Startup.cs文件中注册处理程序和定义策略

```
...
public void ConfigureServices(IServiceCollection servic
es) {

    services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
    services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();
    services.AddSingleton<IClaimsTransformation, Locati
onClaimsProvider>();
    services.AddTransient<IAuthorizationHandler, BlockU
sersHandler>();
    services.AddTransient<IAuthorizationHandler, Docume
ntAuthorizationHandler>();

    services.AddAuthorization(opts => {
        opts.AddPolicy("DCUsers", policy => {
            policy.RequireRole("Users");
            policy.RequireClaim(ClaimTypes.StateOrProvi
nce, "DC");
        });
        opts.AddPolicy("NotBob", policy => {
            policy.RequireAuthenticatedUser();
            policy.AddRequirements(new BlockUsersRequir
ement("Bob"));
        });
        opts.AddPolicy("AuthorsAndEditors", policy => {
            policy.AddRequirements(new DocumentAuthoriz
ationRequirement {
                AllowAuthors = true,
                AllowEditors = true
```

```
            });
        });
    });

    services.AddDbContext<AppIdentityDbContext>(options
 =>
        options.UseSqlServer(
            Configuration["Data:SportStoreIdentity:Conn
ectionString"]));

    services.AddIdentity<AppUser, IdentityRole>(opts =>
 {
        opts.User.RequireUniqueEmail = true;
        //opts.User.AllowedUserNameCharacters = "abcdef
ghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
    }).AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
...
```

最后的步骤是为操作方法应用授权策略，如代码清单30-20所示。

代码清单30-20　在Controllers文件夹下的

DocumentController.cs文件中应用授权策略

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {

    [Authorize]
    public class DocumentController : Controller {
        private ProtectedDocument[] docs = new Protecte
dDocument[] {
            new ProtectedDocument { Title = "Q3 Budget"
, Author = "Alice",
                Editor = "Joe"},
            new ProtectedDocument { Title = "Project Pl
an", Author = "Bob",
                Editor = "Alice"}
        };
        private IAuthorizationService authService;

        public DocumentController(IAuthorizationService
 auth) {
            authService = auth;
        }

        public ViewResult Index() => View(docs);

        public async Task<IActionResult> Edit(string ti
tle) {
            ProtectedDocument doc = docs.FirstOrDefault
(d => d.Title == title);
```

```
            AuthorizationResult authorized = await auth
Service.AuthorizeAsync(User,
                doc, "AuthorsAndEditors");
            if (authorized.Succeeded) {
                return View("Index", doc);
            } else {
                return new ChallengeResult();
            }
        }
    }
}
```

DocumentController控制器类的构造函数定义了IAuthorizationService参数，从而提供了用于评估授权策略的方法，并通过依赖注入提供。在Edit操作方法中，调用AuthorizeAsync方法，传入当前的用户、ProtectedDocument对象以及应用的策略名称。如果AuthorizeAsync方法返回true，授权被批准，然后View方法被调用；如果返回false，就表示存在授权问题，返回一个ChallengeResult对象，如第17章所述，以告诉MVC验证失败。

可以通过执行应用程序并访问URL地

址/Document来查看效果，以不同的用户身份进行验证，例如以Joe身份进行验证，你将能够编辑预算文档，但是不能编辑项目计划文档。

30.4　使用第三方验证

基于声明的系统（如ASP.NET Core Identity）的优势之一是，任何声明都可能来自外部系统，甚至来自识别用户的应用程序。这意味着其他系统可以代表应用程序对用户进行身份验证，ASP.NET Core Identity就建立在这种想法之上，可以简单且容易地添加通过Microsoft、Google、Facebook和Twitter等第三方对用户进行身份验证的功能。

使用第三方验证有很多好处：许多用户已经拥有账户，用户可以选择使用双因子验证，而不必在应用中管理用户凭据。随后的章节将展示如何为Google用户设置和使用第三方身份验证。

30.4.1　注册Google应用

在可以验证用户身份之前，第三方身份验证服务通常需要注册应用程序。注册结果是包含在针对第三方服务的认证请求中的凭据。注册过程可在Google开发人员网站上进行，请按照其中的说明进行操作。必须指定回调地址，默认配置为/signin-google。如果正在开发，可设置回调地址为http://localhost:port/signin-google。对于生产型应用，可创建包含公共主机名和端口的URL地址。

完成注册过程后，你将收到客户端ID，用于将应用程序标识给Google，还将收到客户密钥，以防止其他应用程序伪装成你的应用程序。

注　意

你必须注册自己的应用程序，并使用注册过程生成的客户端标识（客户端 ID）和密钥。除非使用应用程序的唯一值更改凭据，否则代码无法工作。微软提供了名为用户密钥（user secret）的功能，允许你将安全信息存储到外部配置文件中，但为了简单起见，将假设你的配置文件不是共享的，并且可以安全地包含Google身份验证凭据。

30.4.2　启用Google验证

ASP.NET Core Identity内置支持通过Microsoft、Google、Facebook和Twitter账号对用户身份进行验证，还提供对支持OAuth的任意身份验证服务的普遍支持。每种服务都有自己的在Startup类中注册到应用的扩展方法，代码清单30-21展示

了Google服务是如何设置的。为了简洁起见，这里删除了上一示例中的配置语句。

代码清单30-21　在Users文件夹下的Startup.cs文件中启用Google验证

```
...
public void ConfigureServices(IServiceCollection servic
es) {

    services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
    services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();
    services.AddSingleton<IClaimsTransformation, Locati
onClaimsProvider>();
    services.AddTransient<IAuthorizationHandler, BlockU
sersHandler>();
    services.AddTransient<IAuthorizationHandler, Docume
ntAuthorizationHandler>();
    services.AddAuthorization(opts => {
        opts.AddPolicy("DCUsers", policy => {
            policy.RequireRole("Users");
            policy.RequireClaim(ClaimTypes.StateOrProvi
nce, "DC");
        });
        opts.AddPolicy("NotBob", policy => {
            policy.RequireAuthenticatedUser();
            policy.AddRequirements(new BlockUsersRequir
ement("Bob"));
        });
```

```
        opts.AddPolicy("AuthorsAndEditors", policy => {
            policy.AddRequirements(new DocumentAuthoriz
ationRequirement {
                AllowAuthors = true,
                AllowEditors = true
            });
        });
    });

    services.AddAuthentication().AddGoogle(opts => {
        opts.ClientId = "<enter client id here>";
        opts.ClientSecret = "<enter client secret here>
";
    });

    services.AddDbContext<AppIdentityDbContext>(options
 =>
        options.UseSqlServer(
            Configuration["Data:SportStoreIdentity:Conn
ectionString"]));

    services.AddIdentity<AppUser, IdentityRole>(opts =>
 {
        opts.User.RequireUniqueEmail = true;
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
    }).AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
...
```

AddAuthentication.AddGoogle方法设置了使用Google验证用户时必要的服务，以及在注册过程中创建的客户端ID和用户密钥。

当使用第三方验证用户时，可以选择在Identity数据库中创建用户，然后用于管理角色和声明，就像普通用户一样。在第28章中，添加了用户验证类，如果电子邮件地址不在example.com域中，则可以阻止创建用户。由于要处理来自任意域的所有用户，因此必须在该例中禁用电子邮件地址检查，如代码清单30-22所示。

代码清单30-22　在Infrastructure文件夹下的CustomUserValidator.cs文件中禁用验证

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {
```

```
    public class CustomUserValidator : UserValidator<AppUser> {

        public override async Task<IdentityResult> ValidateAsync(
                UserManager<AppUser> manager,
                AppUser user) {

            IdentityResult result = await base.ValidateAsync(manager, user);

            List<IdentityError> errors = result.Succeeded ?
                new List<IdentityError>() : result.Errors.ToList();

            //if (!user.Email.ToLower().EndsWith("@example.com")) {
            //    errors.Add(new IdentityError {
            //        Code = "EmailDomainError",
            //        Description = "Only example.com email addresses are allowed"
            //    });
            //}

            return errors.Count == 0 ? IdentityResult.Success
                : IdentityResult.Failed(errors.ToArray());
        }
    }
}
```

然后在Views/Account/Login.cshtml文件中添加

一个按钮，以允许用户通过Google登录，如代码清单30-23所示。Google为按钮提供了图像，使它们与支持Google账户的其他应用程序保持一致。但为了简单起见，这里仅创建一个标准按钮。

代码清单30-23　在Views/Account文件夹下的Login.cshtml文件中添加按钮

```
@model LoginModel

<div class="bg-primary m-1 p-1 text-white"><h4>Log In</
h4></div>

<div class="text-danger" asp-validation-summary="All"><
/div>

<form asp-action="Login" method="post">
    <input type="hidden" name="returnUrl" value="@ViewB
ag.returnUrl" />
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control"
/>
    </div>
    <button class="btn btn-primary" type="submit">Log I
```

```
n</button>
    <a class="btn btn-info" asp-action="GoogleLogin"
       asp-route-returnUrl="@ViewBag.returnUrl">
        Log In With Google
    </a>
</form>
```

新按钮指向Account控制器的GoogleLogin操作方法。可以看看这个操作方法，以及对控制器所做的其他更改，如代码清单30-24所示。

代码清单30-24　在Controllers文件夹下的AccountController.cs文件中添加对Google验证的支持

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;
using System.Security.Claims;
using Microsoft.AspNetCore.Http.Authentication;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<AppUser> userManager;
        private SignInManager<AppUser> signInManager;

        // ...methods omitted for brevity...
```

```
        [AllowAnonymous]
        public IActionResult GoogleLogin(string returnU
rl) {
            string redirectUrl = Url.Action("GoogleResp
onse", "Account",
                new { ReturnUrl = returnUrl });
            var properties = signInManager
                .ConfigureExternalAuthenticationPropert
ies("Google", redirectUrl);
            return new ChallengeResult("Google", proper
ties);
        }

        [AllowAnonymous]
        public async Task<IActionResult> GoogleResponse
(string returnUrl = "/") {
            ExternalLoginInfo info = await signInManage
r.GetExternalLoginInfoAsync();
            if (info == null) {
                return RedirectToAction(nameof(Login));
            }
            var result = await signInManager.ExternalLo
ginSignInAsync(
                info.LoginProvider, info.ProviderKey, f
alse);
            if (result.Succeeded) {
                return Redirect(returnUrl);
            } else {
                AppUser user = new AppUser {
                    Email = info.Principal.FindFirst(Cl
aimTypes.Email).Value,
                    UserName =
                        info.Principal.FindFirst(ClaimT
ypes.Email).Value
                };
```

```
                IdentityResult identResult = await user
Manager.CreateAsync(user);
                if (identResult.Succeeded) {
                    identResult = await userManager.Add
LoginAsync(user, info);
                    if (identResult.Succeeded) {
                        await signInManager.SignInAsync
(user, false);
                        return Redirect(returnUrl);
                    }
                }
                return AccessDenied();
            }
        }
    }
}
```

GoogleLogin操作方法创建了一个AuthenticationProperties实例，并将RedirectUri属性设置为同一控制器中的GoogleResponse操作方法的URL。以下代码导致ASP.NET Core Identity通过将用户重定向到Google身份验证页面而不是应用程序定义的内容来响应未授权错误：

```
...
return new ChallengeResult("Google", properties);
...
```

这意味着当用户通过单击Log In按钮登录时，他们的浏览器将被重定向到Google验证服务，然后一旦被验证，就被重定向到GoogleResponse操作方法。在GoogleResponse操作方法中，可通过调用SigninManager的GetExternalLoginInfoAsync方法来获取外部登录的详细信息，如下所示：

```
...
ExternalLoginInfo info = await signInManager.GetExternalLoginInfoAsync();
...
```

ExternalLoginInfo类定义了ExternalPrincipal属性，该属性返回一个ClaimsPrincipal对象，该对象包含由Google为用户生成的声明。可使用ExternalLoginSignInAsync方法让用户登录应用程序，如下所示：

```
...
var result = await signInManager.ExternalLoginSignInAsync(
                info.LoginProvider, info.ProviderKey, f
```

```
alse);
...
```

登录失败，这是因为数据库中没有代表Google用户的账户，可创建新用户并将Google账户与之关联，方法是使用如下两条语句：

```
...
IdentityResult identResult = await userManager.CreateAsync(user);
...
identResult = await userManager.AddLoginAsync(user, info);
...
```

注　意

当创建Identity用户时，可使用Google提供的电子邮件声明并用于AppUser对象的Email和UserName

属性，以便不与数据库中现存的任何用户发生命名冲突。

为了测试验证，启动应用程序，单击Log In按钮进行登录，并提供有效的Google账户凭据。完成验证之后，浏览器将被重定向到应用程序。

30.5　小结

本章展示了ASP.NET Core Identity支持的一些高级功能，演示了使用自定义属性以及如何使用数据库迁移来更新数据库模式以执行它们，解释了声明如何工作以及如何使用它们通过策略来创建更灵活的授权用户，还解释了如何使用策略来控制对应用程序管理单个资源的访问，以及如何使用Google验证用户。下一章将介绍如何实现MVC应用程序中

使用的一些最重要的约定，以及如何在自己的应用程序中自定义它们。

第31章　模型约定与操作约束

本章描述两个用于自定义MVC工作方式的有用特性：模型约定允许替换用于创建控制器和操作的默认约定；操作约束允许指定操作可用于何种类型的请求，当MVC选择操作以处理请求时，可提供指导。

表31-1列出了本章要介绍的操作。

表**31-1**　本章要介绍的操作

操作	方法	代码清单
自定义应用模型	使用内置的特性之一或者创建自定义的模型约定	代码清单31-1～代码清单31-14
在整个应用程序中应用自定义约定	定义全局模型约定	代码清单31-15和代码清单31-16
区分可以处理请求的两种	使用action约定	代码清单31-17～代码清

操作方法		单31-25

31.1　准备示例项目

在本章中，使用ASP.NET Core Web Application（.NET Core）模板创建新的名为ConventionsAndConstraints的Empty项目。代码清单31-1展示的Startup类用来设置MVC框架和用于开发的中间件。

代码清单31-1　ConventionsAndConstraints文件夹下的Startup.cs文件的内容

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConventionsAndConstraints {

    public class Startup {
```

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

创建视图模型、控制器和视图

对于本章的大多数示例，知道哪个方法用于响应请求是有帮助的。为此，创建Models文件夹并在其中添加名为Result.cs的类文件，用它定义代码清单31-2所示的类。这个类将允许本章中的控制器将请求是如何处理的信息传递给视图。

代码清单31-2　Models文件夹下的Result.cs文件的内容

```
using System.Collections.Generic;
```

```
namespace ConventionsAndConstraints.Models {

    public class Result {
        public string Controller { get; set; }
        public string Action { get; set; }
    }
}
```

本章只需要一个控制器和视图。创建Controllers文件夹，在其中添加名为HomeController.cs的类文件，并用它定义代码清单31-3所示的类。

代码清单31-3　Controllers文件夹下的HomeController.cs文件的内容

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;

namespace ConventionsAndConstraints.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });
```

```
        public IActionResult List() => View("Result", new Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

Home控制器中的两个操作方法都渲染名为Result的视图，创建Views/Home文件夹，并使用代码清单31-4所示的标记内容在其中创建Result.cshtml视图文件。

代码清单31-4　Views/Home文件夹下的Result.cshtml文件的内容

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
    <title>Result</title>
</head>
```

```
<body class="m-1 p-1">
    <table class="table table-sm table-bordered">
        <tr><th>Controller:</th><td>@Model.Controller</
td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
    </table>
</body>
</html>
```

Result视图基于Bootstrap CSS包为HTML元素设置样式。要将Bootstrap包添加到项目中，可以使用Bower Configuration File模板创建bower.json文件，并将Bootstrap包添加到dependencies部分，如代码清单31-5所示。

代码清单31-5　添加Bootstrap包到ConventionsAndConstraints文件夹下的bower.json文件中

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

最后的准备步骤是在Views文件夹中创建_ViewImports.cshtml文件，从而设置内置的标签助手，将之用于Razor视图并导入Models命名空间，如代码清单31-6所示。

代码清单31-6　Views文件夹下的_ViewImports.cshtml文件的内容

```
@using ConventionsAndConstraints.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

如果运行应用程序，你将看到图31-1所示的结果。

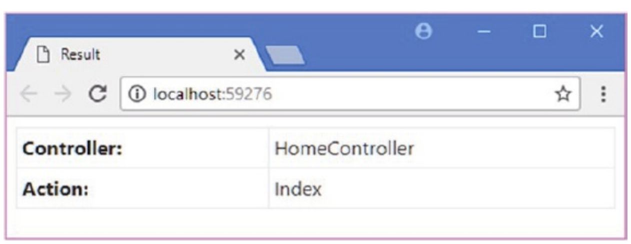


图31-1　运行示例应用程序的结果

31.2　使用应用程序模型和模型约定

MVC偏爱约定胜于配置，因此，可以简单地创建名称以Controller结尾的类，并开始定义操作方法。在运行时，MVC使用发现过程来定位应用中所有的控制器和操作方法，并检查它们是否使用了过滤器等特性。

发现过程的最终结果就是应用程序模型，由描述每个找到的控制器类、操作方法和参数的对象组成。MVC依赖的约定在构建时被应用于应用程序模型。例如，当发现控制器类时，类的名称用于在模型中作为表示控制器的基础；换句话说，HomeController类用来创建名为Home的控制器。当路由系统识别必须由Home控制器处理的请求时，应用程序模型提供到HomeController类的映射。

应用程序模型可以通过模型约定进行定制，模型约定是用于检查应用程序模型内容并进行调整的类，例如合成新的操作方法或者用于创建控制器的

方式等。本节将说明应用程序模型的结构，介绍不同类型的模型约定，并演示模型约定的使用方式。表31-2提供了应用程序模型和模型约定的上下文。

表**31-2**　应用程序模型和模型约定的上下文

问题	答案
它们是什么？	应用程序模型是对应用程序中发现的控制器和操作方法的完整描述。模型约定允许将自定义的更改应用于应用程序模型
它们为什么有用？	模型约定有用是因为它们允许更改将类和方法映射到控制器和操作方法的方式。可以执行其他的定制化操作，例如限制操作可以接收的HTTP方法，或者应用操作约束等（本章稍后介绍）
如何使用它们？	模型约定使用一系列接口来定义，后面将进行说明，可作为特性使用或在Startup类中进行配置
是否有任何陷阱或限制？	在模型约定的应用方式上有一些怪异
有其他备选方案吗？	没有，如果默认的应用程序模型不满足需要，可以通过引入自定义组件来创建自定义的应用程序模型

31.2.1　理解应用程序模型

在发现过程中，MVC创建ApplicationModel类的实例，并使用找到的控制器和操作方法详情进行填充。当发现过程完成之后，应用模型约定以执行指定的任何自定义更新。理解应用程序模型的起点是查看由Microsoft.AspNetCore.Mvc.ApplicationModels.Applic
类定义的属性，如表31-3所示。

表**31-3　ApplicationModel**类定义的属性

属性	描述
Controllers	返回应用程序中包含的所有控制器的IList<ControllerModel>
Filters	返回应用程序中包含的全局过滤器的IList<IFilterMetadata>

注　意

刚开始看起来有些枯燥，特别是当想深入细节时，本节介绍的类完全描述了MVC应用程序的核心部分，值得花点时间来体会。理解应用程序模型的工作原理将有助于你理解更多的高级功能是如何在幕后工作的，当在自己的项目中遇到未预期的问题时，这更有助于诊断问题。

Controllers属性会返回一个包含应用程序中发现的对应每个控制器的ControllerModel对象列表。表31-4介绍了ControllerModel类定义的属性。

表**31-4　ControllerModel**类定义的属性

属性	描述
ControllerName	这个string类型的属性定义了控制器的名称，以匹配controller路由片段中的名称
ControllerType	这个TypeInfo类型的属性定义了控制器的类型
ControllerProperties	返回由控制器定义的描述所有属性的IList<PropertyModel>列表
Actions	返回由控制器定义的描述所有操作方法的IList<ActionModel>列表
Filters	返回控制器中应用于所有操作方法的过滤器的IList<IFilterMetadata>列表
RouteConstraints	返回由控制器定义的如何路由目标操作方法的路由约束的IList<IRouteConstraintProvider>列表
Selectors	返回包含操作方法约束详情的IList<SelectorModel>列表，以及通过特性应用于控制器的路由信息

由此可以看到MVC的一些核心功能是如何被应用程序模型捕获的。例如，ControllerName属性用于设置路由系统用来匹配URL的名称，而ControllerType属性用于设置控制器名称关联的类。

ControllerProperties属性返回一个PropertyModel对象列表，其中的每个PropertyModel对象描述控制器定义的一个属性。表31-5介绍了PropertyModel类定义的属性。

表**31-5　PropertyModel**类定义的属性

属性	描述
PropertyName	返回属性的名称
Attributes	返回应用于属性的特性列表

Actions属性返回一个ActionModel对象列表，其中的每个ActionModel对象描述由单个控制器类定义的一个操作方法。表31-6描述了ActionModel类定义的属性。

表**31-6　ActionModel**类定义的属性

属性	描述

ActionName	这个string类型的属性定义了操作的名称，用于匹配action路由片段
ActionMethod	这个MethodInfo类型的属性用来指定实现操作的方法
Controller	返回用于描述操作所属控制器的ControllerModel对象
Filters	返回应用于操作的过滤器列表IList<IFilterMetadata>
Parameters	返回操作的方法参数描述列表IList<PropertyModel>
RouteConstraints	返回用来限制如何路由到操作的列表IList<IRouteConstraintProvider>
Selectors	返回描述操作约束的列表IList<SelectorModel>，以及通过特性作用于控制器的路由信息

终极的详细信息可通过Parameters属性访问，该属性返回描述操作方法定义的每个参数的ParameterModel对象列表。表31-7描述了ParameterModel类定义的属性。

表**31-7　ParameterModel**类定义的属性

属性	描述

ParameterName	这个string类型的属性表示参数名称
ParameterInfo	表示指定的参数信息
BindingInfo	用于配置模型绑定过程

ApplicationModel、ControllerModel、PropertyModel、ActionModel和ParameterModel用于描述应用程序中控制器类的各个方面，比如方法、属性以及每个方法定义的参数。

自定义应用程序模型

MVC有一些内置的约定，可使用ControllerModel、PropertyModel、ActionModel和ParameterModel对象来填充ApplicationModel，以描述发现的控制器。

一些约定是显式的，例如将Controller从控制器的类名中移除，用来设置ControllerModel对象的

ControllerName属性。这个约定意味着定义名为HomeController的类，但在URL片段中使用Home来定位。

其他约定是隐式的，例如类用来创建控制器，方法用来创建操作。大多数MVC开发人员将这些约定视为理所当然，但是实际上，应用程序模型的各个方面都可以改变。

前面的章节描述了用来改变MVC工作方式的特性，它们实际上是模型约定。表31-8介绍了这些特性。

表**31-8**　用来改变默认应用程序约定的一些基本特性

特性	描述
ActionName	允许显式指定ActionModel的ActionName属性，而不是从方法名中派生
NonController	防止类被用于创建ControllerModel对象

NonAction	防止方法被用于创建ActionModel对象

代码清单31-7使用ActionName特性修改了创建于HomeController类中的List方法的操作名称。

代码清单31-7　在Controllers文件夹下的HomeController.cs中定制应用程序模型

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;

namespace ConventionsAndConstraints.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionName("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

名称Details将被用于创建操作，并替换默认的名称List。可以通过启动应用程序并访问/Home/Details来查看效果，如图 31-2所示，请求被List方法处理。

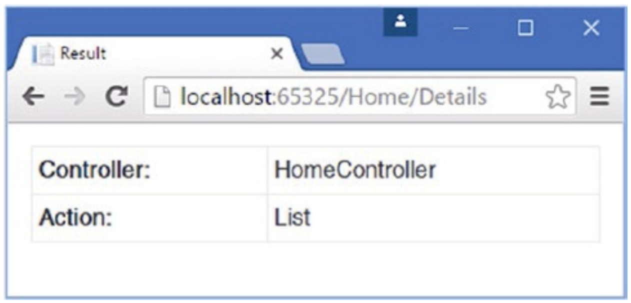


图31-2　定制应用程序模型

31.2.2　理解模型约定角色

表31-8描述的特性允许对应用程序模型进行基本的变更，但是范围有限。对于更实质的定制，需要使用模型约定。

来自表31-8的特性允许在应用程序模型对象创建之前指定变更，例如覆盖操作的名称。相比之

下，模型约定允许在模型对象创建之后更新应用程序模型，从而允许进行更广泛的更新。有4种类型的模型约定，每种通过不同的接口进行定义，如表31-9所示。

表**31-9**　应用程序模型约定接口

接口	描述
IApplicationModelConvention	应用模型约定到ApplicationModel对象
IControllerModelConvention	应用模型约定到应用程序模型的ControllerModel对象
IActionModelConvention	应用模型约定到应用程序模型的ActionModel对象
IParameterModelConvention	应用模型约定到应用程序模型的ParameterModel对象

这4种接口都以相同的方式工作，只是它们修改的应用程序模型层面不同。例如，下面是IControllerModel- Convention接口的定义：

```
namespace Microsoft.AspNetCore.Mvc.ApplicationModels {
```

```
    public interface IControllerModelConvention {

        void Apply(ControllerModel controller);
    }
}
```

通过调用Apply方法可提供对应用程序模型约定的目标修改ControllerModel的机会，作为方法的参数接收。其他接口也定义了Apply方法，并且每个都接收对应类型的模型对象，比如IActionModelConvention接口接收ActionModel对象，而IParameterModelConvention接口接收ParameterModel对象。

31.2.3 创建模型约定

控制器、操作和参数模型约定可以作为特性应用，这样可以很容易设置应用更改的范围。作为演示，创建Infrastructure文件夹并添加名为ActionNamePrefixAttribute.cs的类文件，用它定义代

码清单31-8所示的类。

代码清单31-8　Infrastructure文件夹下的ActionNamePrefixAttribute.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace ConventionsAndConstraints.Infrastructure {

    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public class ActionNamePrefixAttribute : Attribute, IActionModelConvention {
        private string namePrefix;

        public ActionNamePrefixAttribute(string prefix) {
            namePrefix = prefix;
        }

        public void Apply(ActionModel action) {
            action.ActionName = namePrefix + action.ActionName;
        }
    }
}
```

ActionNamePrefixAttribute类派生自Attribute类并实现了IActionModelConvention接口。

ActionNamePrefix- Attribute类的构造函数接受用作前缀的字符串，该字符串用于修改Apply方法接收的ActionModel对象的ActionName属性。

提　示

这里限制了ActionNamePrefix特性的使用，它只能应用于方法。当以特性的方式应用模型约定时，控制器约定只有当作用于类时才能生效，操作约定只有当作用于方法时才能生效，参数约定只有当作用于参数才能生效。在错误的级别应用约定会被简单地忽略而不产生任何错误。为避免混淆，可使用AttributeUsage来限制创建的特性。

代码清单31-9将模型约定应用到了Home控制器的操作方法上。

代码清单31-9　在Controllers文件夹下的HomeController.cs文件中应用模型约定

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionNamePrefix("Do")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

当MVC执行发现过程的时候，将创建用于描

述List方法的ActionModel对象，检测ActionNamePrefix并调用Apply方法。可以通过运行应用程序并访问/Home/DoList来查看效果，在默认约定下，定位到List方法的URL已经被替换了，如图31-3所示。

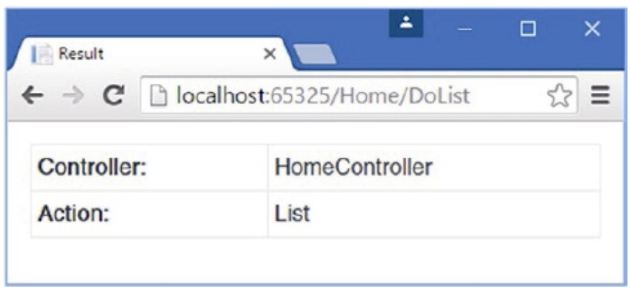


图31-3　应用模型约定

使用约定添加或删除模型

使用模型约定的一种奇怪方式是阻止在应用程序模型中添加或删除对象。例如，假设要创建一个约定，可以通过两种不同的操作来访问某些方法。为了演示，在Infrastructure文件夹中创建名为AddActionAttribute.cs的类文件，并用它定义代码清

单31-10所示的类。

代码清单31-10　Infrastructure文件夹下的AddActionAttribute.cs文件的内容

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace ConventionsAndConstraints.Infrastructure {

    [AttributeUsage(AttributeTargets.Method, AllowMulti
ple = true)]
    public class AddActionAttribute : Attribute, IActio
nModelConvention {
        private string additionalName;

        public AddActionAttribute(string name) {
            additionalName = name;
        }

        public void Apply(ActionModel action) {
            action.Controller.Actions.Add(new ActionMod
el(action) {
                ActionName = additionalName
            });
        }
    }
}
```

这个模型约定使用ActionModel构造函数来复

制现有对象并修改新对象的ActionName属性。可通过ActionModel.Controller属性将新的ActionModel添加到控制器的操作集合中。在代码清单31-11中，可以看到如何将模型约定应用于Home控制器。

代码清单31-11　在Controllers文件夹下的HomeController.cs文件中应用模型约定

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
```

```
}
```

启动应用程序，MVC将开始发现过程并报告如下错误：

```
InvalidOperationException: Collection was modified; enu
meration operation may not execute.
```

当操作集合被发现过程枚举时，模型约定试图修改action模型对象的集合，从而导致异常。为避免错误，可采用不同的方法，如代码清单31-12所示。

代码清单31-12　在Infrastructure文件夹下的AddActionAttribute.cs文件中创建安全的模型约定

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace ConventionsAndConstraints.Infrastructure {

    [AttributeUsage(AttributeTargets.Method, AllowMulti
ple = true)]
    public class AddActionAttribute : Attribute {

```

```
        public string AdditionalName { get; }

        public AddActionAttribute(string name) {
            AdditionalName = name;
        }
    }

    [AttributeUsage(AttributeTargets.Class, AllowMultip
le = false)]
    public class AdditionalActionsAttribute : Attribute,

            IControllerModelConvention {

        public void Apply(ControllerModel controller) {
            var actions = controller.Actions
                .Select(a => new {
                    Action = a,
                    Names = a.Attributes.Select(attr =>
                        (attr as AddActionAttribute)?.A
dditionalName)
                });

            foreach (var item in actions.ToList()) {
                foreach (string name in item.Names) {
                    controller.Actions.Add(new ActionMo
del(item.Action) {
                        ActionName = name
                    });
                }
            }
        }
    }
}
```

在action模型约定中，不能修改控制器相关的操作方法集合。但是，仍然需要以某种方式表达要做的修改。因此，将AddActionAttribute作为特性而不是模型约定使用。

你可以在控制器的模型约定中修改操作方法集合，这就是为什么创建AdditionalActionsAttribute类。Apply方法使用LINQ来定位使用了AddActionAttribute特性的方法并使用指定的名称创建新的ActionModel对象。

对于AdditionalActionsAttribute类，最重要的部分是在LINQ查询结果上调用ToList方法：

```
...
foreach (var item in actions.ToList()) {
...
```

ToList方法强制对LINQ查询结果求值并用结果生成一个新的集合。这意味着foreach循环将在另外

的对象集上枚举。若没有ToList调用，就会收到与代码清单31-12相同的错误；若使用ToList调用，就能创建新的action模型对象。代码清单31-13展示了如何将修改后的特性应用于Home控制器。

代码清单31-13　在Controllers文件夹下的HomeController.cs文件中应用修改后的模型约定

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    [AdditionalActions]
    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
```

```
    }
}
```

可以通过启动应用程序并访问/Home/Details和/Home/List来查看修改后的模型约定效果。如图31-4所示，模型约定添加了由List方法处理的新操作，补充了默认创建的操作模型。

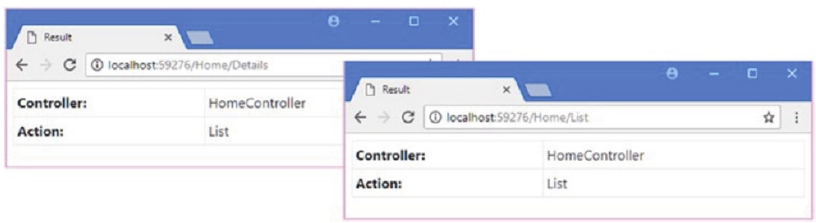


图31-4　创建新的操作模型

31.2.4　理解模型约定的执行顺序

模型约定将以特定的次序应用，从最大范围开始：首先应用控制器模型约定，然后应用操作模型约定，最后应用参数模型约定。为了演示，将前面几个示例中创建的自定义模型约定应用于HomeController类的List方法，如代码清单31-14所

示。

代码清单31-14　在Controllers文件夹下的HomeController.cs文件中应用多个模型约定

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    [AdditionalActions]
    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionNamePrefix("Do")]
        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

首先应用作为控制器模型约定的

AdditionalActions特性，并创建名为Details的操作。然后，应用作为操作模型约定的ActionNamePrefix，将Do前缀应用于操作方法关联的操作。结果导致List方法实现了两个操作——DoList和DoDetails，它们可以使用URL地址/Home/DoList和/Home/DoDetails访问，如图31-5所示。

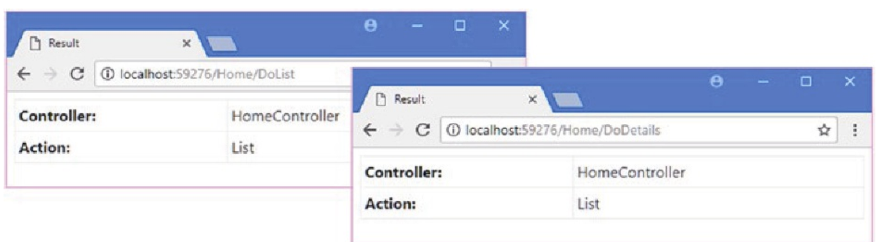


图31-5　多个模型约定的执行效果

31.2.5　创建全局模型约定

如果需要更改默认模型约定，那么可能必须为应用程序中的每个控制器、操作或参数执行更改操作。如果是这种情况，那么可以创建全局模型约

定，而不必将特性一一应用于每个控制器类。全局模型约定在Startup类中配置，如代码清单31-15所示。

代码清单31-15　在ConventionsAndConstraints文件夹下的Startup.cs文件中创建全局过滤器

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc().AddMvcOptions(options => {

                options.Conventions.Add(new ActionNameP
refixAttribute("Do"));
                options.Conventions.Add(new AdditionalA
ctionsAttribute());
            });
```

```
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

由AddMvcOptions扩展方法接收的MvcOptions对象定义了Conventions属性。这个属性将返回用于添加模型约定对象的列表集合，其中的列表可全局应用两个自定义模型约定。这意味着所有的操作名称将以Do为前缀，并将检查所有的AddAction特性。由于这些模型约定将被全局应用，因此从HomeController类中将它们删除，如代码清单31-16所示。

代码清单31-16　从Controllers文件夹下的HomeController.cs文件中删除模型约定

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    //[AdditionalActions]
    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        //[ActionNamePrefix("Do")]
        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

在模型约定作用于类之前，应用全局模型约定。如果有多个全局模型约定，那么它们将按照注册的顺序应用，而不考虑类型。在控制器模型约定之前注册操作模型约定，这意味着在将

ActionNamePrefixAttribute约定应用于所有操作之后，通过AddAction特性指定的Details操作才被创建。因此，List方法实现了两个操作——DoList和Details，它们可以通过URL地址/Home/DoList和/Home/Details进行访问，如图31-6所示。

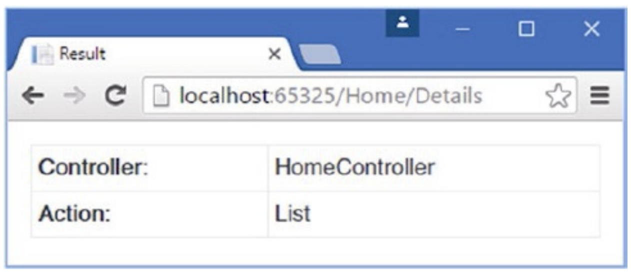


图31-6　全局模型约定的执行结果

31.3　使用操作约束

操作约束决定了操作方法是否适合用来处理特定请求，这可能导致你认为操作约束类似于第19章描述的授权过滤器。

实际上，操作约束更受限。当MVC接收到

HTTP请求时，将通过一个选择过程来识别用于处理HTTP请求的操作方法。如果有多个操作方法可以处理请求，那么MVC需要使用一些手段来决定使用哪个操作方法，这就是使用操作约束的原因。表31-10提供操作约束的背景。

表**31-10**　操作约束的背景

问题	答案
它们是什么？	操作约束是MVC用来确定HTTP请求是否可以由特定操作方法处理的类
它们为什么有用？	如果有两个或更多个操作方法可以处理某个请求，那么MVC需要使用一些手段来决定使用哪个操作方法，操作约束用来提供这些信息
如何使用？	操作约束通过特性来使用，这允许它们在整个应用程序中重用，且意味着确定操作是否应该处理请求的逻辑不必在操作方法中定义
有什么陷阱或限制吗？	操作约束可以被广泛应用，并阻止请求被任何适当的操作方法处理，导致向客户端发送无用的404 - Not Found响应
有备选方案吗？	如果需要在特定情况下限制对操作的访问，那么过滤器更为有用，可以重定向客户端以便显示有用的错误页面

31.3.1　准备示例项目

操作约束的目的是帮助MVC在两个或多个可以用于处理请求的操作方法之间进行选择。代码清单31-17在Home控制器中添加了另一个新的操作方法。

代码清单31-17　在Controllers文件夹下的HomeController.cs文件中创建两个合适的操作方法

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    //[AdditionalActions]
    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionName("Index")]
        public IActionResult Other() => View("Result",
new Result {
```

```
            Controller = nameof(HomeController),
            Action = nameof(Other)
        });

        //[ActionNamePrefix("Do")]
        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

以上代码添加了一个新的名为Other的操作方法并应用ActionName特性以便处理为名为Index的操作。更新Startup类以删除本章前面的全局模型约定，如代码清单31-18所示。

代码清单31-18　在ConventionsAndConstraints文件夹下的Startup.cs文件中删除全局模型约定

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {

    public class Startup {
        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddMvc().AddMvcOptions(options =>
{
                //options.Conventions.Add(new ActionNam
ePrefixAttribute("Do"));
                //options.Conventions.Add(new Additiona
lActionsAttribute());
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

这意味着Home控制器中存在两个名为Index的操作，如果运行应用，将看到图31-7所示的错误消息，指出MVC不知道应用使用哪个操作方法。

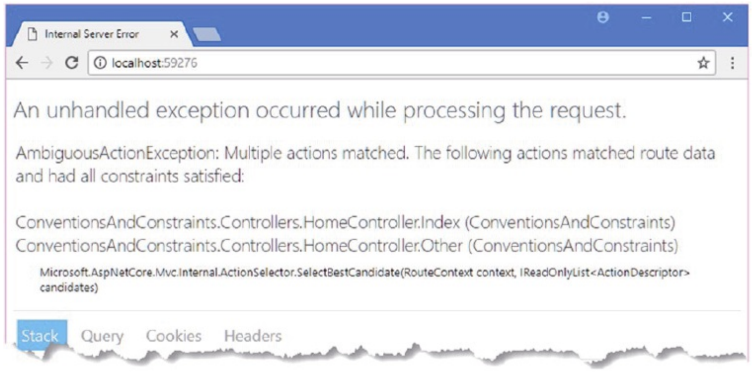


图31-7　错误消息

以上是错误信息的相关内容：

```
AmbiguousActionException: Multiple actions matched. The
 following actions matched route
data and had all constraints satisfied:
ConventionsAndConstraints.Controllers.HomeController.In
dex
ConventionsAndConstraints.Controllers.HomeController.Ot
her
```

31.3.2　操作约束的作用

操作约束用来告诉MVC是否可以使用操作方法处理请求和实现IActionConstraint接口，这个接口的定义如下：

```
namespace Microsoft.AspNetCore.Mvc.ActionConstraints {
```

```
    public interface IActionConstraint : IActionConstraintMetadata {

        int Order { get; }

        bool Accept(ActionConstraintContext context);
    }
}
```

MVC在经过选择操作方法来处理请求的过程后，将检查是否存在与之关联的约束。如果有，它们将按照Order属性的值依次组织，并且依次调用每个约束的Accept方法。只要任何约束的Accept方法返回false，MVC就知道操作方法不能用于处理当前请求。

提　示

IActionConstraint接口派生自IActionConstraintMetadata，后者是没有定义成员的接口，并不直接使用。当创建自定义约束时，应当使用IActionConstraint接口；当期望创建有依赖要处理的约束时，应当使用IActionConstraintFactory接口。

为了有助于操作约束做出决定，MVC提供了ActionConstraintContext实例作为上下文数据，Action- ConstraintContext类定义的属性如表31-11所示。

表**31-11　ActionConstraintContext**类定义的属性

属性	描述
Candidates	返回ActionSelectorCandidate对象列表，描述MVC用于处理当前请求的候选操作方法列表

CurrentCandidate	返回ActionSelectorCandidate对象，描述约束当前正在请求评估的操作方法
RouteContext	返回RouteContext对象，提供路由数据（通过RouteData属性）以及HTTP请求（通过HttpContext属性）

31.3.3　创建操作约束

最常见的操作约束是确认请求符合某些策略，比如提供特定的HTTP请求头。为了演示此种类型的操作约束，在示例项目的Infrastructure文件夹中添加名为UserAgentAttribute.cs的类文件，用它定义代码清单31-19所示的类。

代码清单31-19　Infrastructure文件夹夹下的UserAgentAttribute.cs文件的内容

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;

namespace ConventionsAndConstraints.Infrastructure {

    public class UserAgentAttribute : Attribute, IActio
nConstraint {
```

```
        private string substring;

        public UserAgentAttribute(string sub) {
            substring = sub.ToLower();
        }

        public int Order { get; set; } = 0;

        public bool Accept(ActionConstraintContext cont
ext) {
            return context.RouteContext.HttpContext
                .Request.Headers["User-Agent"]
                .Any(h => h.ToLower().Contains(substrin
g));
        }
    }
}
```

以上操作约束用于在UserAgent请求头不包含特定字符串的时候，阻止匹配的操作方法处理请求。在Accept方法中，从HttpContext对象中获取请求头，然后使用LINQ来检查其中之一是否包含通过构造函数接收的子串。

注　意

在实际的应用程序中不要基于User-Agent头来识别浏览器，因为请求头的值经常会误导人。例如，在编写本书时，Microsoft Edge浏览器发送的User-Agent请求头中包含了Android、Apple、Chrome和Safari，使得很容易误认为其他的浏览器。更稳健的方式是在应用中使用JavaScript库（比如Modernizr）来检测浏览器。

代码清单31-20为HomeController类中的一个方法应用了操作约束。

代码清单31-20　在Controllers文件夹下的HomeController.cs文件中应用操作约束

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    //[AdditionalActions]
    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionName("Index")]
        [UserAgent("Edge")]
        public IActionResult Other() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Other)
        });

        //[ActionNamePrefix("Do")]
        [AddAction("Details")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

为Other操作方法应用这个特性，并指定如果

请求的User-Agent请求头中没有包含Edge，就不接收请求。

启动应用程序，分别使用Google Chrome和Microsoft Edge浏览器访问/Home/Index，你将看到不同的处理结果，如图31-8所示。

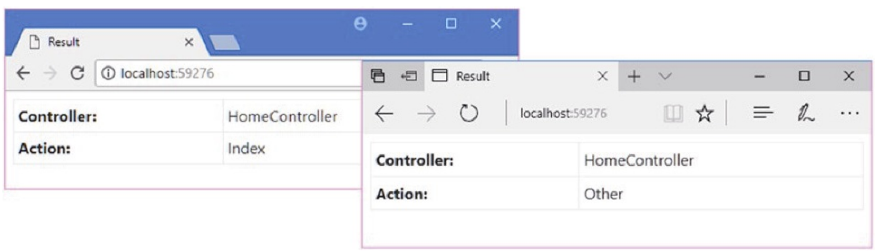


图31-8　操作约束的使用效果

理解操作方法的约束效果

前面的示例揭示了使用操作约束的如下方面：对于给定的请求，带有Accept方法且返回true的应用了操作约束的操作方法优先于没有应用操作约束

的操作方法。

Home控制器中有两个名为Index的操作，分别由Index和Other操作方法创建，它们都可以处理带有请求头User-Agent且值为Edge的请求。Other操作方法用来处理来自Edge浏览器的请求这是由于为之应用了操作约束，并且Accept方法返回true。理念就是，带接收请求的操作约束的操作方法总是优先于没有操作约束的操作方法。

创建比较用的操作约束

通过ActionConstraintContext对象的Candidates和CurrentCandidate属性，操作约束提供了处理请求的其他候选操作方法的详情。每个潜在的匹配可使用ActionSelectorCandidate实例来描述，用到的属性

如表31-12所示。

表**31-12　ActionSelectorCandidate**类定义的属性

属性	描述
Action	返回描述候选操作方法的ActionDescriptor对象
Constraints	返回包含应用到候选操作方法的IActionConstraint约束列表

ActionDescriptor类用于通过表31-13所示的属性描述操作方法，与其他上下文对象提供的内容类似。

表**31-13　ActionDescriptor**类定义的属性

属性	描述
Name	返回操作方法的名称
RouteConstraints	返回用来限制如何路由到操作方法的IList<IRouteConstraintProvider>列表
Parameters	返回操作方法参数描述的IList<PropertyModel>列表

ActionConstraints	返回操作方法的约束列表IList<IActionConstraintMetadata>

操作约束可以检查候选的操作方法并洞察如何以及在哪里应用，还可以微调它们如何工作。作为示例，考虑代码清单31-21中应用到Home控制器的操作约束。

代码清单31-21　在Controllers文件夹下的HomeController.cs文件中为控制器应用操作约束

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() => View("Result",
new Result {
            Controller = nameof(HomeController),
            Action = nameof(Index)
        });

        [ActionName("Index")]
        [UserAgent("Edge")]
        public IActionResult Other() => View("Result",
new Result {
```

```
            Controller = nameof(HomeController),
            Action = nameof(Other)
        });

        [UserAgent("Edge")]
        public IActionResult List() => View("Result", n
ew Result {
            Controller = nameof(HomeController),
            Action = nameof(List)
        });
    }
}
```

这里仅为名为List的操作方法应用了操作约束，这意味着仅在请求头User-Agent中包含Edge的请求可使用这个操作方法处理。例如，如果使用Google Chrome发出请求，将收到404 - Not Found响应。

这没什么用，因为用户不知道为什么收到错误，也没有说明性的文本来建议使用其他的浏览器替代Google Chrome。在控制处理请求的操作方法的选择时，操作约束很有用；使用过滤器将允许把客户重定向到有错误说明的页面，这才是更实用的

响应。

为了处理这个问题，更新UserAgentAttribute类，以便在只有一个候选操作方法的时候不拒绝请求，如代码清单31-22所示。

代码清单31-22　在Infrastructure文件夹下的UserAgentAttribute.cs文件中检查其他候选操作方法

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;
namespace ConventionsAndConstraints.Infrastructure {

    public class UserAgentAttribute : Attribute, IActio
nConstraint {
        private string substring;

        public UserAgentAttribute(string sub) {
            substring = sub.ToLower();
        }

        public int Order { get; set; } = 0;

        public bool Accept(ActionConstraintContext cont
ext) {
            return context.RouteContext.HttpContext
                    .Request.Headers["User-Agent"]
                    .Any(h => h.ToLower().Contains(subs
```

```
tring))
                    || context.Candidates.Count() == 1;
        }
    }
}
```

附加的LINQ查询将检查CurrentCandidate返回的候选操作方法是否为Candidates属性返回的集合中的唯一元素。如果是，那么操作约束就知道MVC没有其他的候选操作方法而允许请求。使用Google Chrome浏览器运行应用程序并请求URL地址/Home/List来查看效果。即使通过Google Chrome发送请求的User-Agent请求头中没有包含Edge，约束类也将因为检测到没有其他的候选操作方法而允许请求被处理。

31.3.4　在操作约束中处理依赖

当需要通过service provider在操作约束中处理依赖的时候，使用IActionConstraintFactory接口。如第18章所示，下面是这个接口的定义：

```
using System;

namespace Microsoft.AspNetCore.Mvc.ActionConstraints {

    public interface IActionConstraintFactory : IAction
ConstraintMetadata {

        IActionConstraint CreateInstance(IServiceProvid
er services);

        bool IsReusable { get; }
    }
}
```

CreateInstance方法将被调用以创建操作约束，IsReusable属性用来指示通过CreateInstance方法创建的对象是否可以用于多个请求。

为了演示这个接口的使用，要用到依赖关系。为此，在Infrastructure文件夹中添加名为UserAgentComparer.cs的类文件，并用它定义代码清单31-23所示的类。

代码清单31-23　Infrastructure文件夹下的UserAgentComparer.cs文件的内容

```
using System.Linq;
using Microsoft.AspNetCore.Http;

namespace ConventionsAndConstraints.Infrastructure {

    public class UserAgentComparer {

        public bool ContainsString(HttpRequest request,
 string agent) {
            string searchTerm = agent.ToLower();
            return request.Headers["User-Agent"]
                .Any(h => h.ToLower().Contains(searchTe
rm));
        }
    }
}
```

UserAgentComparer类定义了一个用于在User-Agent请求头中查询字符串的方法。功能与前面的示例相同，但是打包成单独的类，以便可以使用service provider来管理生命周期，如代码清单31-24所示。

代码清单31-24　在ConventionsAndConstraints文件夹下的Startup.cs文件中为service provider注册类型

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {

    public class Startup {

        public void ConfigureServices(IServiceCollectio
n services) {
            services.AddSingleton<UserAgentComparer>();
            services.AddMvc().AddMvcOptions(options =>
{
                //options.Conventions.Add(new ActionNam
ePrefixAttribute("Do"));
                //options.Conventions.Add(new Additiona
lActionsAttribute());
            });
        }

        public void Configure(IApplicationBuilder app,
IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

以上代码选择了单例生命周期，这意味着UserAgentComparer类只有单个实例。代码清单31-25更新了UserAgent约束以便将检查请求头的任务委托给通过service provider获取的UserAgentComparer对象。

代码清单31-25　在Infrastructure文件夹下的UserAgentAttribute.cs文件中解决依赖问题

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;
using Microsoft.Extensions.DependencyInjection;

namespace ConventionsAndConstraints.Infrastructure {

    public class UserAgentAttribute : Attribute, IActio
nConstraintFactory {
        private string substring;

        public UserAgentAttribute(string sub) {
            substring = sub;
        }

        public IActionConstraint CreateInstance(IServic
eProvider services) {
            return new UserAgentConstraint(services.Get
Service<UserAgentComparer>(),
```

```
                substring);
        }

        public bool IsReusable => false;

        private class UserAgentConstraint : IActionCons
traint {
            private UserAgentComparer comparer;
            private string substring;

            public UserAgentConstraint(UserAgentCompare
r comp, string sub) {
                comparer = comp;
                substring = sub.ToLower();
            }

            public int Order { get; set; } = 0;

            public bool Accept(ActionConstraintContext
context) {
                return comparer.ContainsString(context.
RouteContext
                        .HttpContext.Request, substring
)
                    || context.Candidates.Count() == 1;
            }
        }
    }
}
```

在以上模型中，应用于操作方法的特性负责在调用CreateInstance方法时创建约束类的实例。

CreateInstance方法的参数是一个IserviceProvider对象，本例使用了UserAgentComparer，以便可以创建私有约束类的实例，然后在选择过程中使用。

避免范围陷阱

与其他基于特性的功能一样，将约束特性应用于控制器类等同于将约束特性应用于每个独立的方法。然而，这通常会导致未预期的结果，因为操作约束的目的是帮助MVC选择操作方法，而不是对控制器中所有的操作方法应用相同的约束。

例如，如果将UserAgent特性应用于HomeController类，那么任何浏览器都将无法访问Index这个操作方法。两个Index操作方法都同样适用于处理User-Agent字符串中包含Edge的浏览器，这将导致异常。对于所有其他的浏览器，这两个

Index操作方法都不合适，这将导致404-Not Found响应。

使用约束中的上下文对象可以查找其他的约束，并查看它们是否可能拒绝请求，但这会导致每个约束的Accept方法被调用多次。当有多个操作方法可以处理相同的请求时，可将约束应用于这些操作方法，此时最有效。

31.4　小结

本章描述了两个用于定制MVC运行方式的特性，解释了如何使用模型约束来改变将类和方法映射到控制器和操作的方式，还讲述了如何使用操作约束来限制操作方法可能处理的请求的范围，以及如何使用它们在请求到达时从识别的候选列表中选

择一个操作方法。

本书从创建一个简单的应用程序开始，带你全面了解了ASP.NET Core MVC框架中不同的组件，理解如何配置、定制或完全替换它们。

最后，祝你在MVC项目中取得圆满成功！